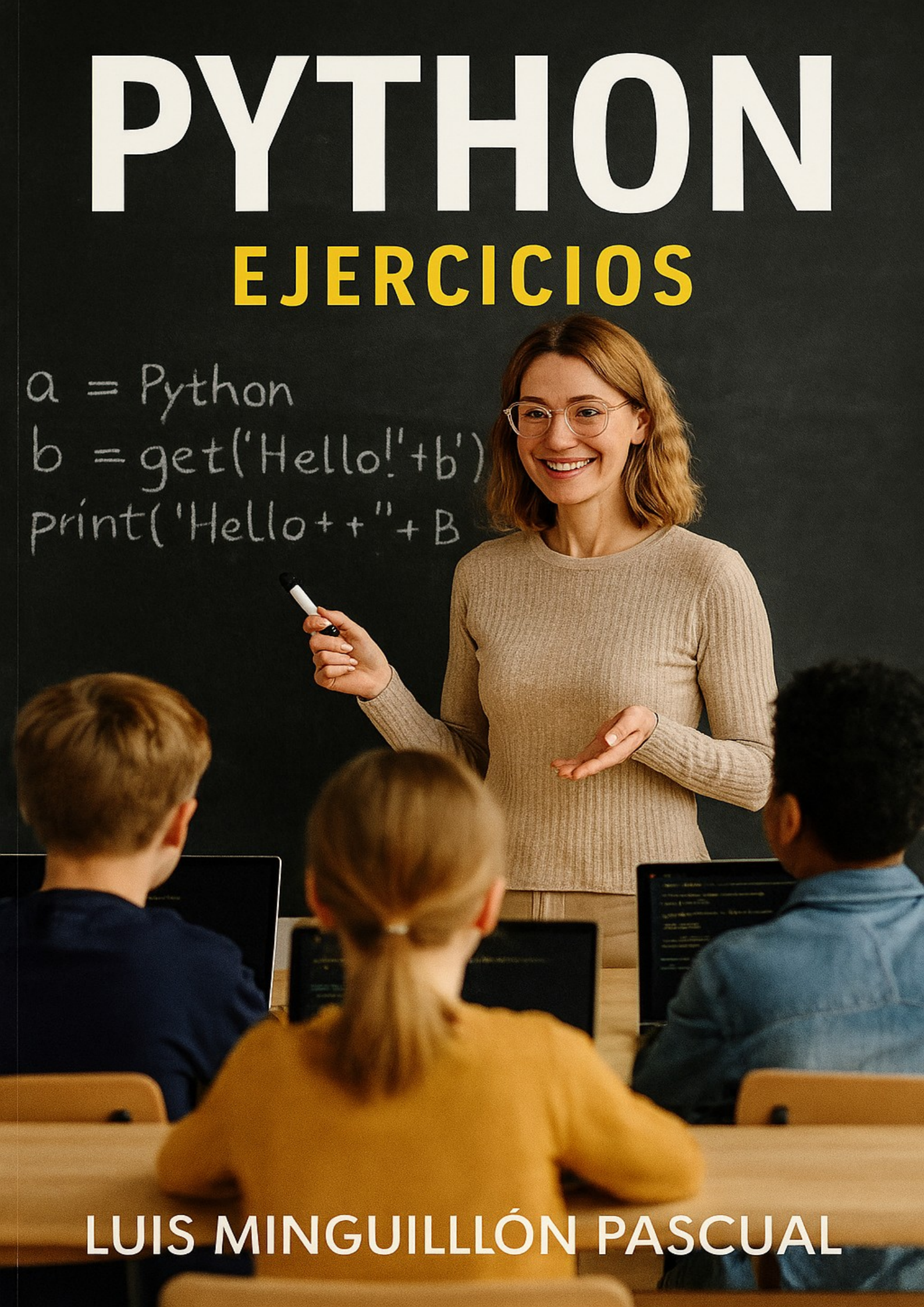
PYTHON

EJERCICIOS



LUIS MINGUILLÓN PASCUAL

Python
ejercicios

Luis Minguillón Pascual

COPYRIGHT

La licencia de este libro electrónico es para uso personal por lo tanto no puedes revenderlo a otras personas.

Gracias por respetar los derechos de autor.

©Luis Minguillón Pascual
16/19/2025

Prefacio

El presente libro de ejercicios de programación en Python está diseñado con el objetivo de acompañar al lector en un proceso de aprendizaje práctico, progresivo y estructurado. He decidido dividir el contenido en **19 secciones**, y cada una de ellas se compone de dos partes: un pequeño manual introductorio que explica los conceptos fundamentales de la sección y una colección de ejercicios destinados a reforzar lo aprendido.

Esta organización responde a una idea muy clara: facilitar la comprensión de cada tema antes de pasar a la práctica. De esta manera, el lector no solo encontrará problemas para resolver, sino también una base teórica que lo guiará y le dará las herramientas necesarias para enfrentarlos con confianza.

Los ejercicios están pensados para **abarcar un recorrido que va desde un nivel principiante hasta un nivel avanzado**, de modo que cualquier persona, sin importar su experiencia previa, pueda progresar de manera ordenada en el dominio del lenguaje Python. Considero que la mejor forma de aprender a programar es practicando, y este libro ha sido creado bajo esa convicción: que la práctica constante es el verdadero camino hacia la comprensión y la destreza.

No quiero dejar pasar la oportunidad de recordar a mi mujer, **Marisa**, porque sin su apoyo, su paciencia y su confianza en mí, este proyecto no habría sido posible. A ella dedico este esfuerzo, que espero sea también de ayuda e inspiración para todos aquellos que deseen aprender a programar.

Luis Minguillón Pascual

MANUAL DETALLADO DE EJERCICIOS DE PYTHON

BLOQUE I: Variables y Operadores Básicos (Ejercicios 1 - 7)

Ejercicio	Enunciado y Objetivo	Explicación del Proceso (Controles y Comandos)
1	**Suma de dos números:** Escribe un programa que pida al usuario dos números y muestre la suma de ambos.	El comando **`input()`** se utiliza para pedir información al usuario. El comando **`int()`** convierte el valor ingresado a número entero, asegurando que se pueda operar matemáticamente. El operador **`+`** realiza la suma. Finalmente, **`print()`** muestra el resultado en pantalla. Las variables utilizadas son `num1`, `num2` y `suma`.
2	**Área de un rectángulo:** Calcula el área de un rectángulo a partir de su base y altura.	Se usa la función **`float()`** para convertir los valores ingresados (base y altura) a números decimales, permitiendo entradas no enteras. El operador **`*`** realiza la multiplicación de la base por la altura. Las variables principales son `base`, `altura` y `area`.
3	**Promedio de tres números:** Calcula el promedio de tres números ingresados por el usuario.	Se ingresan tres números usando **`float()`**. Se suman los tres números y se dividen entre 3. El operador **`/`** realiza la división flotante. Variables clave: `n1`, `n2`, `n3`, `promedio`.
4	**Número par o impar:** Determina si un número ingresado es par o impar.	El operador **`%` (módulo)** calcula el residuo de la división del número entre 2. La estructura condicional **`if`** evalúa si este residuo es cero (indicando par). El operador **`==`** compara la igualdad. Si la condición es falsa, se ejecuta el bloque **`else`**.
5	**Tabla de multiplicar:** Muestra la tabla de multiplicar de un número ingresado.	El bucle **`for`** se utiliza para recorrer un rango (del 1 al 10 en este caso). La función **`range()`** genera esta secuencia. El operador **`*`** multiplica el número ingresado (`n`) por el contador del bucle (`i`).
6	**Conversión de grados Celsius a Fahrenheit:** Convierte una temperatura en grados Celsius a Fahrenheit.	La conversión se basa en la fórmula $F = C * 9/5 + 32$. Se emplean los operadores **`*`** (multiplicación), **`/`** (división) y **`+`** (suma). Se usa **`float()`** para manejar la temperatura como número decimal.
7	**Número positivo, negativo o cero:** Determina si un número ingresado es positivo, negativo o cero.	Las estructuras condicionales **`if`**, **`elif`** (que permite múltiples condiciones) y **`else`** se emplean para verificar el signo. Los operadores de comparación **`>`** (mayor que) y **`<`** (menor que) se usan para determinar si el número es positivo o negativo, respectivamente.

BLOQUE II: Bucles, Listas y Acumulación (Ejercicios 8 - 15)

Ejercicio	Enunciado y Objetivo	Explicación del Proceso (Controles y Comandos)
8	**Factorial de un número:** Calcula el factorial de un número ingresado por el usuario.	El bucle **`for`** recorre desde 1 hasta el número ingresado. El operador **`*=`** multiplica el valor actual de la variable `factorial` por el contador del bucle (`i`) y asigna el resultado de nuevo a `factorial`.
9	**Números pares entre 1 y 20:** Muestra todos los números pares del 1 al 20.	**`range(1, 21)`** se usa para recorrer la secuencia del 1 al 20. Dentro del bucle, la condición **`if i % 2 == 0`** detecta los números pares.
10	**Suma de números pares hasta n:** Suma todos los números pares desde 1 hasta n.	Similar al E9, se utiliza **`for`** e **`if`** para filtrar solo los números pares. El operador **`+=`** se utiliza para sumar y asignar el valor de cada par a la variable `suma`.
11	**Suma de elementos de una lista:** Suma todos los números de una lista dada por el usuario.	La función **`split(",")`** divide la cadena de texto ingresada (separada por comas) en elementos de una lista. El bucle **`for`** recorre esta lista, y la función **`int()`** convierte cada elemento a entero antes de sumarlo usando **`+=`**.
12	**Máximo de una lista:** Encuentra el número más grande en una lista ingresada por el usuario.	Se utiliza **`split(",")`** para obtener los elementos. La sintaxis de **List Comprehension** (`[int(n) for n in numeros]`) convierte todos los elementos de la lista a enteros. La función **`max()`** se usa directamente para devolver el mayor elemento de la lista.
13	**Números primos hasta n:** Muestra todos los números primos desde 2 hasta n.	Se usa un **bucle anidado**; el bucle externo recorre de 2 a n, y el bucle interno verifica la divisibilidad. Si se encuentra un divisor (`i % j == 0`), la instrucción **`break`** detiene el bucle interno, ya que el número no es primo.
14	**Contar elementos en una lista:** Cuenta cuántos números hay en una lista ingresada por el usuario.	Después de usar **`split(",")`** para generar la lista, un bucle **`for`** itera sobre ella. La expresión **`contador += 1`** incrementa el contador por cada elemento que se encuentra.
15	**Invertir una lista:** Invierte el orden de los elementos de una lista ingresada.	Se utiliza **`split(",")`** para crear la lista. La inversión se logra de manera eficiente utilizando **slicing** con la sintaxis **`[::-1]`**.

BLOQUE III: Funciones, Cadenas y Colecciones (Ejercicios 16 - 25)

Ejercicio	Enunciado y Objetivo	Explicación del Proceso (Controles y Comandos)
16	**Promedio de lista usando función:** Crea una función que calcule el promedio de una lista de números.	La palabra clave **`def`** se utiliza para definir la función. Se retorna el resultado de dividir (operador **`/`**) la suma de los elementos (**`sum(lista)`**) entre el tamaño de la lista (**`len(lista)`**).
17	**Número de vocales en una palabra:** Cuenta cuántas vocales tiene una palabra ingresada.	La función **`.lower()`** convierte la entrada a minúsculas para simplificar la comparación. El bucle **`for`** itera sobre cada letra. La estructura **`if letra in vocales`** verifica si la letra actual está contenida en la cadena de vocales.
18	**Contar ocurrencias de un elemento:** Cuenta cuántas veces aparece un elemento en una lista.	Se utiliza el método de lista **`.count(elemento)`**, que devuelve directamente el número de veces que aparece un valor específico en la lista.
19	**Lista de números pares:** Crea una lista con todos los números pares hasta n.	Se utiliza **List Comprehension** para generar la lista en una sola línea. Se emplea la condición **`if i % 2 == 0`** para filtrar solo los números pares.
20	**Números cuadrados:** Muestra los cuadrados de los números del 1 al 10.	Se usa el bucle **`for`** junto con **`range(1, 11)`**. El operador de potencia **`**`** se usa para elevar el número al cuadrado (`i**2`).
21	**Contar palabras en una frase:** Cuenta cuántas palabras hay en una frase ingresada por el usuario.	El método **`.split()`** sin argumentos divide la frase por espacios y devuelve una lista de palabras. La función **`len()`** devuelve el número de elementos de la lista, que corresponde al conteo de palabras.
22	**Reemplazar vocales en una cadena:** Reemplaza todas las vocales de una palabra por el carácter `*`.	Se recorre cada letra con **`for`**. Dentro del **`if`**, se verifica si la letra (convertida a minúscula con **`.lower()`**) está contenida (**`in`**) en "aeiou". El operador **`+=`** se usa para concatenar el carácter adecuado al resultado.
23	**Diccionario de edades:** Muestra la edad de un nombre ingresado desde un diccionario.	La instrucción **`if nombre in edades`** verifica si la clave (nombre) existe en el diccionario. **`edades[nombre]`** se usa para obtener el valor (la edad) asociado a esa clave. Se utiliza **`if / else`** para manejar el caso de nombres no encontrados.
24	**Suma de valores en un diccionario:** Suma todas las edades de un diccionario.	El método **`.values()`** devuelve una vista de los valores contenidos en el diccionario. La función **`sum()`** suma todos estos valores.
25	**Palíndromo:** Verifica si una palabra es un palíndromo.	La palabra se convierte a minúsculas con **`.lower()`**. Se invierte la cadena usando **slicing `[::-1]`**. El operador **`==`** compara si la palabra original es igual a la invertida.

BLOQUE IV: Listas Avanzadas y Flujo de Control (Ejercicios 26 - 30)

Ejercicio	Enunciado y Objetivo	Explicación del Proceso (Controles y Comandos)
26	**Números impares en una lista:** Genera una lista con todos los números impares hasta n.	Se emplea **List Comprehension**. La condición utilizada para el filtrado es **`if i % 2 != 0`**, donde **`!=`** significa "no es igual".
27	**Promedio de números pares:** Calcula el promedio solo de los números pares de una lista.	Se utiliza **List Comprehension** (`[n for n in numeros if n % 2 == 0]`) para primero filtrar y crear una lista solo con los números pares. Luego se calcula el promedio dividiendo la suma (**`sum()`**) entre la longitud (**`len()`**) de la nueva lista de pares.
28	**Fibonacci hasta n:** Muestra la serie de Fibonacci hasta un número n.	Se utiliza el bucle **`while`**, que se repite mientras la condición **`a <= n`** sea verdadera. La **asignación simultánea `a, b = b, a + b`** actualiza los dos valores siguientes de la serie en un solo paso.
29	**Contar letras en una frase:** Cuenta cuántas veces aparece cada letra en una frase.	Se usa **`.lower()`** para estandarizar el texto. El bucle **`for`** itera sobre la frase. **`.isalpha()`** verifica si el carácter es una letra. El método de diccionario **`.get(letra, 0)`** obtiene el conteo actual de la letra (o `0` si es la primera vez que aparece) antes de sumarle 1.
30	**Ordenar lista de números:** Ordena una lista de números de menor a mayor.	Se utiliza el método de lista **`.sort()`**, que ordena la lista en el lugar (modificando la lista original directamente).

BLOQUE V: Archivos y Manejo de Errores (Ejercicios 31 - 33)

Ejercicio	Enunciado y Objetivo	Explicación del Proceso (Controles y Comandos)
31	**Escribir en un archivo:** Escribe una frase ingresada por el usuario en un archivo.	La estructura **`with open("archivo.txt", "w") as archivo:`** abre el archivo en modo escritura (`"w"`). El comando **`with`** asegura que el archivo se cierre automáticamente. El método **`.write(frase)`** escribe el texto en el archivo.
32	**Leer de un archivo:** Lee y muestra el contenido de un archivo.	La estructura **`with open("archivo.txt", "r")`** abre el archivo en modo lectura (`"r"`). El método **`.read()`** obtiene todo el contenido del archivo en una sola cadena.
33	**Manejo de excepciones:** Pide un número y maneja la excepción si ingresa un valor no válido.	La estructura **`try`** envuelve el código que podría fallar (la conversión `int(input(...))` si el usuario ingresa texto). La estructura **`except ValueError:`** captura específicamente la excepción generada cuando la conversión a entero falla, permitiendo mostrar un mensaje de error.

BLOQUE VI: Funciones Avanzadas (Ejercicios 34 - 37)

Ejercicio	Enunciado y Objetivo	Explicación del Proceso (Controles y Comandos)
34	**Función para sumar dos números:** Crea una función que reciba dos números y devuelva su suma.	Se define la función con **`def sumar(a, b):`**. La palabra clave **`return`** se usa para devolver el resultado de la suma (operador **`+`**).
35	**Función que verifica par:** Crea una función que devuelva `True` si un número es par y `False` si es impar.	La función utiliza la expresión **`return num % 2 == 0`**. Esta única línea de código evalúa si el número es divisible entre 2 y devuelve el resultado booleano directamente.
36	**Lista de cuadrados con función:** Crea una función que devuelva una lista con los cuadrados de 1 a n.	La función utiliza **List Comprehension** para generar la lista de forma compacta: `[i**2 for i in range(1, n+1)]`. El operador **`**`** eleva cada número al cuadrado.
37	**Números primos con función:** Crea una función que devuelva todos los números primos hasta n.	La función utiliza bucles anidados para verificar la primalidad. Si se determina que un número es primo, el método de lista **`.append(i)`** agrega ese número a la lista `primos`.

BLOQUE VII: Manipulación de Colecciones y Diccionarios (Ejercicios 38 - 49)

Ejercicio	Enunciado y Objetivo	Explicación del Proceso (Controles y Comandos)
38	**Contar palabras repetidas:** Cuenta cuántas veces se repite cada palabra en una frase.	La frase se procesa usando **`.lower().split()`** para obtener una lista de palabras en minúsculas. Para el conteo, se utiliza **`.get(palabra, 0)`**, que obtiene el conteo actual o devuelve 0 si la palabra es nueva, y luego le suma 1 (operador **`+`**).
39	**Ordenar diccionario por valor:** Ordena un diccionario por sus valores.	**`.items()`** devuelve tuplas de (clave, valor). La función **`sorted()`** ordena estas tuplas. Se usa la expresión **`key=lambda x: x`** para indicar que la ordenación debe realizarse basándose en el segundo elemento de la tupla (el valor). Finalmente, **`dict()`** convierte el resultado ordenado de nuevo a diccionario.
40	**Contar números negativos y positivos:** Cuenta cuántos números negativos y positivos hay en una lista.	Se utiliza **List Comprehension** anidada dentro de la función **`sum()`**. Esto permite contar directamente cuántos elementos cumplen la condición (`n > 0` o `n < 0`).
41	**Guardar lista en archivo:** Guarda los elementos de una lista en un archivo, uno por línea.	Se recorre la lista con un bucle **`for`**. El método **`.write()`** escribe cada `item` en el archivo, y se le concatena (`+`) el salto de línea **`\n`**.
42	**Leer archivo línea por línea:** Lee un archivo y muestra cada línea.	La sintaxis **`for linea in archivo`** recorre directamente el archivo línea por línea. El método **`.strip()`** es crucial para eliminar los saltos de línea y los espacios en blanco que pueda haber al inicio o al final de cada línea leída.
43	**Validar entrada de número:** Pide un número hasta que se ingrese uno válido.	El bucle **`while True`** crea un ciclo que se ejecuta indefinidamente hasta que se rompe. El bloque **`try / except ValueError`** intenta convertir la entrada a entero. Si tiene éxito, **`break`** se ejecuta para romper el bucle.
44	**Lista de diccionarios:** Muestra solo los mayores de 18 en una lista de diccionarios.	Se utiliza **List Comprehension** para filtrar la lista de diccionarios. La condición **`if p["edad"] > 18`** filtra los diccionarios según el valor de la clave "edad".
45	**Sumar valores de diccionario:** Suma los valores de un diccionario de ventas.	Se usan los métodos **`.values()`** para obtener las cantidades vendidas y **`sum()`** para totalizarlas.
46	**Crear diccionario de frecuencias:** Cuenta la frecuencia de cada número en una lista.	Se utiliza un bucle **`for`** y el método de diccionario **`.get(n, 0)`**. Este método es clave, ya que obtiene el valor de la clave `n` si existe, o devuelve 0 para inicializar el conteo si es la primera vez que se encuentra el número.
47	**Función para buscar en lista:** Busca un elemento y devuelve su índice o -1.	La estructura **`if valor in lista`** verifica si el elemento está presente. Si está, el método **`lista.index(valor)`** devuelve su posición; si no, devuelve -1.

48	**Merge de diccionarios:** Combina dos diccionarios en uno solo.	El método `.update(d2)` agrega todos los elementos del diccionario `d2` al diccionario `d1`. Si hay claves repetidas (como "b"), `d1` toma el valor de `d2`.
49	**Eliminar duplicados de lista:** Crea una lista sin elementos duplicados.	La función **`set()`** elimina automáticamente los elementos duplicados al crear un conjunto. La función **`list()`** convierte el conjunto resultante de nuevo a una lista para el resultado final.

BLOQUE VIII: Procesamiento de Archivos y Cálculos Finales (Ejercicio 50)

Ejercicio	Enunciado y Objetivo	Explicación del Proceso (Controles y Comandos)
50	**Promedio de edades desde archivo:** Lee edades desde un archivo y calcula el promedio.	Se abre el archivo usando **`with open(...)`**. Se utiliza **List Comprehension** para leer cada línea (`linea`), eliminar espacios/saltos de línea (**`.strip()`**), y convertirla a entero (**`int()`**). Finalmente, se calcula el promedio dividiendo la suma (**`sum()`**) de las edades entre la cantidad de edades (**`len()`**).

Ejercicios de Python, Variables y operadores

Ejercicio 1: Suma de dos números

Enunciado:
Escribe un programa que pida al usuario dos números y muestre la suma de ambos.

Código:

```
# Pedimos al usuario dos números
num1 = int(input("Ingresa el primer número: "))
num2 = int(input("Ingresa el segundo número: "))

# Calculamos la suma
suma = num1 + num2

# Mostramos el resultado
print("La suma es:", suma)
```

Explicación del código:

1. `input()` pide información al usuario.
2. `int()` convierte el valor ingresado a número entero.
3. `+` es el operador de suma.
4. `print()` muestra información en pantalla.

Explicación de los comandos:

- **Variables (`num1`, `num2`, `suma`)**: almacenan valores.
- **Operadores (`+`)**: realizan operaciones matemáticas.
- **Funciones (`input()`, `int()`, `print()`)**: interactúan con el usuario y procesan datos.

Ejercicio 2: Área de un rectángulo

Enunciado:
Calcula el área de un rectángulo a partir de su base y altura.

Código:

```
base = float(input("Ingrese la base: "))
altura = float(input("Ingrese la altura: "))

area = base * altura

print("El área del rectángulo es:", area)
```

Explicación del código:

- `float()` convierte el valor a número decimal.
- `*` multiplica base por altura.
- `print()` muestra el resultado.

Comandos:

- Variables: `base`, `altura`, `area`.
- Operadores: `*` (multiplicación).
- Funciones: `input()`, `float()`, `print()`.

Ejercicio 3: Promedio de tres números

Enunciado:
Calcula el promedio de tres números ingresados por el usuario.

Código:

```
n1 = float(input("Número 1: "))
n2 = float(input("Número 2: "))
n3 = float(input("Número 3: "))

promedio = (n1 + n2 + n3) / 3

print("El promedio es:", promedio)
```

Explicación:

- Se suman los tres números y se dividen entre 3 (`/`).
- `/` realiza la división flotante.

Comandos:

- Variables: `n1`, `n2`, `n3`, `promedio`.
- Operadores: `+`, `/`.
- Funciones: `input()`, `float()`, `print()`.

Ejercicio 4: Número par o impar

Enunciado:
Determina si un número ingresado es par o impar.

Código:

```
numero = int(input("Ingrese un número: "))

if numero % 2 == 0:
    print("El número es par")
else:
    print("El número es impar")
```

Explicación:

- `%` calcula el residuo de la división.
- `if` evalúa la condición.
- `==` compara igualdad.

Comandos:

- Variables: `numero`.
- Operadores: `%` (módulo), `==` (igualdad).
- Estructura condicional: `if` / `else`.

Ejercicio 5: Tabla de multiplicar

Enunciado:
Muestra la tabla de multiplicar de un número ingresado.

Código:

```
n = int(input("Ingrese un número: "))

for i in range(1, 11):
    print(n, "x", i, "=", n * i)
```

Explicación:

- `for` crea un bucle que recorre del 1 al 10.
- `*` multiplica el número por el contador.

Comandos:

- Variable: `n`, `i`.
- Operadores: `*`.
- Funciones: `input()`, `int()`, `print()`, `range()`.
- Bucle: `for`.

Ejercicio 6: Conversión de grados Celsius a Fahrenheit

Enunciado:
Convierte una temperatura en grados Celsius a Fahrenheit.

Código:

```
celsius = float(input("Ingrese la temperatura en Celsius: "))

fahrenheit = (celsius * 9/5) + 32

print("La temperatura en Fahrenheit es:", fahrenheit)
```

Explicación:

- La fórmula es `F = C * 9/5 + 32`.
- `*` y `/` son operadores de multiplicación y división.

Comandos:

- Variables: `celsius`, `fahrenheit`.
- Operadores: `*`, `/`, `+`.
- Funciones: `input()`, `float()`, `print()`.

Ejercicio 7: Número positivo, negativo o cero

Enunciado:

Determina si un número ingresado es positivo, negativo o cero.

Código:

```
num = float(input("Ingrese un número: "))

if num > 0:
    print("Positivo")
elif num < 0:
    print("Negativo")
else:
    print("Cero")
```

Explicación:

- `elif` permite múltiples condiciones.
- `>`, `<` son operadores de comparación.

Comandos:

- Variables: `num`.
- Operadores: `>`, `<`.
- Estructuras condicionales: `if`, `elif`, `else`.

Ejercicio 8: Factorial de un número

Enunciado:
Calcula el factorial de un número ingresado por el usuario.

Código:

```
num = int(input("Ingrese un número: "))
factorial = 1

for i in range(1, num + 1):
    factorial *= i

print("El factorial es:", factorial)
```

Explicación:

- `*=` multiplica y asigna.
- Bucle `for` recorre del 1 al número ingresado.

Comandos:

- Variables: `num`, `factorial`, `i`.
- Operadores: `*=`.
- Funciones: `input()`, `int()`, `print()`, `range()`.
- Bucle: `for`.

Ejercicio 9: Números pares entre 1 y 20

Enunciado:
Muestra todos los números pares del 1 al 20.

Código:

```
for i in range(1, 21):
    if i % 2 == 0:
        print(i)
```

Explicación:

- `% 2` detecta números pares.
- `range(1, 21)` recorre del 1 al 20.

Comandos:

- Variable: `i`.
- Operadores: `%`, `==`.
- Estructura condicional: `if`.
- Bucle: `for`.
- Función: `print()`.

Ejercicio 10: Suma de números pares hasta n

Enunciado:

Suma todos los números pares desde 1 hasta n.

Código:

```
n = int(input("Ingrese un número: "))
suma = 0

for i in range(1, n + 1):
    if i % 2 == 0:
        suma += i

print("La suma de los pares hasta", n, "es:", suma)
```

Explicación:

- `+=` suma y asigna.
- Se usan `for` y `if` para filtrar los pares.

Comandos:

- Variables: `n`, `suma`, `i`.
- Operadores: `%`, `+=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `for`, `if`.

Ejercicio 11: Suma de elementos de una lista

Enunciado:
Crea un programa que sume todos los números de una lista dada por el usuario.

Código:

```
numeros = input("Ingrese números separados por comas: ").split(",")
suma = 0

for n in numeros:
    suma += int(n)

print("La suma es:", suma)
```

Explicación:

- `split(",")` divide el texto ingresado en elementos de una lista.
- `for` recorre la lista y suma cada elemento convertido a entero.

Comandos:

- Variables: `numeros`, `suma`, `n`.
- Operadores: `+=`.
- Funciones: `input()`, `split()`, `int()`, `print()`.
- Bucle: `for`.

Ejercicio 12: Máximo de una lista

Enunciado:
Encuentra el número más grande en una lista ingresada por el usuario.

Código:

```
numeros = input("Ingrese números separados por comas: ").split(",")
numeros = [int(n) for n in numeros]

maximo = max(numeros)

print("El número máximo es:", maximo)
```

Explicación:

- `[int(n) for n in numeros]` convierte todos los elementos a enteros.
- `max()` devuelve el mayor elemento de la lista.

Comandos:

- Variables: `numeros`, `maximo`.
- Funciones: `input()`, `split()`, `int()`, `max()`, `print()`.
- Operadores: ninguno específico.

Ejercicio 13: Números primos hasta n

Enunciado:
Muestra todos los números primos desde 2 hasta n.

Código:

```
n = int(input("Ingrese un número: "))

for i in range(2, n + 1):
    primo = True
    for j in range(2, i):
        if i % j == 0:
            primo = False
            break
    if primo:
        print(i)
```

Explicación:

- Bucle anidado: el externo recorre 2 a n, el interno verifica divisibilidad.
- `break` detiene el bucle si encontramos un divisor.

Comandos:

- Variables: `n`, `i`, `j`, `primo`.
- Operadores: `%`, `=`, `==`.
- Estructuras: `for`, `if`, `break`.
- Funciones: `input()`, `int()`, `print()`.

Ejercicio 14: Contar elementos en una lista

Enunciado:

Cuenta cuántos números hay en una lista ingresada por el usuario.

Código:

```
numeros = input("Ingrese números separados por comas: ").split(",")
contador = 0

for n in numeros:
    contador += 1

print("Cantidad de números:", contador)
```

Explicación:

- `contador += 1` suma 1 por cada elemento en la lista.

Comandos:

- Variables: `numeros`, `contador`, `n`.
- Operadores: `+=`.
- Bucle: `for`.
- Funciones: `input()`, `split()`, `print()`.

Ejercicio 15: Invertir una lista

Enunciado:
Invierte el orden de los elementos de una lista ingresada.

Código:

```
lista = input("Ingrese elementos separados por comas: ").split(",")
lista_invertida = lista[::-1]

print("Lista invertida:", lista_invertida)
```

Explicación:

- `[::-1]` invierte la lista usando slicing.

Comandos:

- Variables: `lista`, `lista_invertida`.
- Operadores: `:` (slicing).
- Funciones: `input()`, `split()`, `print()`.

Ejercicio 16: Promedio de lista usando función

Enunciado:
Crea una función que calcule el promedio de una lista de números.

Código:

```
def promedio(lista):
    return sum(lista) / len(lista)

numeros = [2, 4, 6, 8]
print("El promedio es:", promedio(numeros))
```

Explicación:

- `def` define una función.
- `sum(lista)` suma elementos.
- `len(lista)` devuelve el tamaño de la lista.

Comandos:

- Variables: `lista`, `numeros`.
- Funciones: `def`, `sum()`, `len()`, `print()`.
- Operadores: `/`.

Ejercicio 17: Número de vocales en una palabra

Enunciado:

Cuenta cuántas vocales tiene una palabra ingresada.

Código:

```
palabra = input("Ingrese una palabra: ").lower()
vocales = "aeiou"
contador = 0

for letra in palabra:
    if letra in vocales:
        contador += 1

print("Cantidad de vocales:", contador)
```

Explicación:

- `.lower()` convierte a minúsculas.
- `in` verifica si la letra es vocal.

Comandos:

- Variables: `palabra`, `vocales`, `contador`, `letra`.
- Operadores: `+=`.
- Estructura: `for`, `if`, `in`.
- Funciones: `input()`, `lower()`, `print()`.

Ejercicio 18: Contar ocurrencias de un elemento

Enunciado:

Cuenta cuántas veces aparece un elemento en una lista.

Código:

```
lista = [1, 2, 2, 3, 4, 2, 5]
elemento = 2
contador = lista.count(elemento)

print("El elemento", elemento, "aparece", contador, "veces")
```

Explicación:

- `.count()` devuelve cuántas veces aparece un valor en la lista.

Comandos:

- Variables: `lista`, `elemento`, `contador`.
- Funciones: `count()`, `print()`.

Ejercicio 19: Lista de números pares

Enunciado:
Crea una lista con todos los números pares hasta n.

Código:

```
n = int(input("Ingrese un número: "))
pares = [i for i in range(2, n+1) if i % 2 == 0]

print("Números pares:", pares)
```

Explicación:

- List comprehension genera la lista en una sola línea.

Comandos:

- Variables: `n`, `pares`, `i`.
- Operadores: %.
- Funciones: `input()`, `int()`, `range()`, `print()`.
- Estructuras: `if` dentro de list comprehension.

Ejercicio 20: Números cuadrados

Enunciado:
Muestra los cuadrados de los números del 1 al 10.

Código:

```
for i in range(1, 11):
    print("El cuadrado de", i, "es", i**2)
```

Explicación:

- `**` es el operador de potencia.

Comandos:

- Variables: `i`.
- Operadores: `**`.
- Funciones: `range()`, `print()`.
- Bucle: `for`.

Ejercicio 21: Contar palabras en una frase

Enunciado:
Cuenta cuántas palabras hay en una frase ingresada por el usuario.

Código:

```
frase = input("Ingrese una frase: ")
palabras = frase.split()
cantidad = len(palabras)
print("Número de palabras:", cantidad)
```

Explicación:

- `split()` sin argumentos divide la frase por espacios.
- `len()` devuelve el número de elementos de la lista.

Comandos:

- Variables: `frase`, `palabras`, `cantidad`.
- Funciones: `input()`, `split()`, `len()`, `print()`.

Ejercicio 22: Reemplazar vocales en una cadena

Enunciado:
Reemplaza todas las vocales de una palabra por el carácter `*`.

Código:

```
palabra = input("Ingrese una palabra: ")
resultado = ""

for letra in palabra:
    if letra.lower() in "aeiou":
        resultado += "*"
    else:
        resultado += letra

print("Resultado:", resultado)
```

Explicación:

- Recorremos cada letra y usamos `in` para verificar si es vocal.
- `+=` concatena el carácter adecuado a la cadena resultado.

Comandos:

- Variables: `palabra`, `resultado`, `letra`.
- Operadores: `+=`.
- Funciones: `input()`, `lower()`, `print()`.
- Estructuras: `for`, `if`.

Ejercicio 23: Diccionario de edades

Enunciado:
Crea un diccionario con nombres y edades, y muestra la edad de un nombre ingresado.

Código:

```
edades = {"Ana": 20, "Luis": 25, "Marta": 30}
nombre = input("Ingrese un nombre: ")

if nombre in edades:
    print(nombre, "tiene", edades[nombre], "años")
else:
    print("Nombre no encontrado")
```

Explicación:

- `in` verifica si la clave está en el diccionario.
- `edades[nombre]` obtiene el valor asociado a la clave.

Comandos:

- Variables: `edades`, `nombre`.
- Operadores: `in`.
- Funciones: `input()`, `print()`.
- Estructuras: `if` / `else`.

Ejercicio 24: Suma de valores en un diccionario

Enunciado:
Suma todas las edades de un diccionario.

Código:

```
edades = {"Ana": 20, "Luis": 25, "Marta": 30}
total = sum(edades.values())
print("Suma de edades:", total)
```

Explicación:

- `.values()` devuelve los valores del diccionario.
- `sum()` los suma todos.

Comandos:

- Variables: `edades`, `total`.
- Funciones: `values()`, `sum()`, `print()`.

Ejercicio 25: Palíndromo

Enunciado:
Verifica si una palabra es un palíndromo (se lee igual al derecho y al revés).

Código:

```
palabra = input("Ingrese una palabra: ").lower()
if palabra == palabra[::-1]:
    print("Es un palíndromo")
else:
    print("No es un palíndromo")
```

Explicación:

- `[::-1]` invierte la cadena.
- `==` compara si la palabra original es igual a la invertida.

Comandos:

- Variables: `palabra`.
- Operadores: `==`.
- Funciones: `input()`, `lower()`, `print()`.

Ejercicio 26: Números impares en una lista

Enunciado:
Genera una lista con todos los números impares hasta n.

Código:

```
n = int(input("Ingrese un número: "))
impares = [i for i in range(1, n+1) if i % 2 != 0]
print("Números impares:", impares)
```

Explicación:

- List comprehension con condición `i % 2 != 0`.

Comandos:

- Variables: `n`, `impares`, `i`.
- Operadores: `%`, `!=`.
- Funciones: `input()`, `int()`, `range()`, `print()`.

Ejercicio 27: Promedio de números pares

Enunciado:
Calcula el promedio solo de los números pares de una lista.

Código:

```
numeros = [1, 2, 3, 4, 5, 6]
pares = [n for n in numeros if n % 2 == 0]
promedio = sum(pares) / len(pares)
print("Promedio de pares:", promedio)
```

Explicación:

- Filtramos los pares con list comprehension y luego calculamos el promedio.

Comandos:

- Variables: `numeros`, `pares`, `promedio`, `n`.
- Operadores: `%`, `==`, `/`.
- Funciones: `sum()`, `len()`, `print()`.

Ejercicio 28: Fibonacci hasta n

Enunciado:
Muestra la serie de Fibonacci hasta un número n.

Código:

```
n = int(input("Ingrese un número: "))
a, b = 0, 1

while a <= n:
    print(a, end=" ")
    a, b = b, a + b
```

Explicación:

- `while` se repite mientras la condición sea verdadera.
- `a, b = b, a + b` actualiza los valores simultáneamente.

Comandos:

- Variables: `n`, `a`, `b`.
- Operadores: `+`, `<=`.
- Funciones: `input()`, `int()`, `print()`.
- Bucle: `while`.

Ejercicio 29: Contar letras en una frase

Enunciado:
Cuenta cuántas veces aparece cada letra en una frase.

Código:

```
frase = input("Ingrese una frase: ").lower()
contador = {}

for letra in frase:
    if letra.isalpha():
        contador[letra] = contador.get(letra, 0) + 1

print(contador)
```

Explicación:

- `isalpha()` verifica si es letra.
- `.get(letra, 0)` obtiene valor actual o 0 si no existe la clave.

Comandos:

- Variables: `frase`, `contador`, `letra`.
- Operadores: `+`.
- Funciones: `input()`, `lower()`, `isalpha()`, `get()`, `print()`.
- Estructuras: `for`, `if`.

Ejercicio 30: Ordenar lista de números

Enunciado:
Ordena una lista de números de menor a mayor.

Código:

```
numeros = [5, 2, 9, 1, 7]

numeros.sort()

print("Lista ordenada:", numeros)
```

Explicación:

- `.sort()` ordena la lista en el lugar (modifica la lista original).

Comandos:

- Variables: `numeros`.
- Funciones: `sort()`, `print()`.

Ejercicio 31: Escribir en un archivo

Enunciado:
Crea un programa que escriba una frase ingresada por el usuario en un archivo llamado `archivo.txt`.

Código:

```
frase = input("Ingrese una frase: ")

with open("archivo.txt", "w") as archivo:
    archivo.write(frase)

print("Frase guardada en archivo.txt")
```

Explicación:

- `open("archivo.txt", "w")` abre o crea un archivo para escritura.
- `write()` escribe la frase en el archivo.
- `with` asegura cerrar el archivo automáticamente.

Comandos:

- Variables: `frase`, `archivo`.
- Funciones: `input()`, `open()`, `write()`, `print()`.
- Estructura: `with`.

Ejercicio 32: Leer de un archivo

Enunciado:
Lee y muestra el contenido de un archivo llamado `archivo.txt`.

Código:

```
with open("archivo.txt", "r") as archivo:
    contenido = archivo.read()

print("Contenido del archivo:", contenido)
```

Explicación:

- `"r"` indica modo lectura.
- `read()` obtiene todo el contenido del archivo.

Comandos:

- Variables: `archivo`, `contenido`.
- Funciones: `open()`, `read()`, `print()`.
- Estructura: `with`.

Ejercicio 33: Manejo de excepciones

Enunciado:
Pide al usuario un número y maneja la excepción si ingresa un valor no válido.

Código:

```
try:
    num = int(input("Ingrese un número: "))
    print("Número ingresado:", num)
except ValueError:
    print("¡Eso no es un número válido!")
```

Explicación:

- `try` ejecuta código que puede fallar.
- `except` captura la excepción `ValueError`.

Comandos:

- Variables: `num`.
- Operadores: ninguno.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `try` / `except`.

Ejercicio 34: Función para sumar dos números

Enunciado:
Crea una función que reciba dos números y devuelva su suma.

Código:

```
def sumar(a, b):
    return a + b

resultado = sumar(5, 7)
print("La suma es:", resultado)
```

Explicación:

- `def` define la función.
- `return` devuelve el resultado.

Comandos:

- Variables: `a`, `b`, `resultado`.
- Operadores: `+`.
- Funciones: `def`, `return`, `print()`.

Ejercicio 35: Función que verifica par

Enunciado:
Crea una función que devuelva `True` si un número es par y `False` si es impar.

Código:

```
def es_par(num):
    return num % 2 == 0


print(es_par(4))
print(es_par(7))
```

Explicación:

- `% 2 == 0` determina si el número es divisible entre 2.

Comandos:

- Variables: `num`.
- Operadores: `%`, `==`.
- Funciones: `def`, `return`, `print()`.

Ejercicio 36: Lista de cuadrados con función

Enunciado:
Crea una función que devuelva una lista con los cuadrados de los números de 1 a n.

Código:

```
def cuadrados(n):
    return [i**2 for i in range(1, n+1)]


print(cuadrados(5))
```

Explicación:

- List comprehension genera la lista de cuadrados.
- `**2` eleva al cuadrado cada número.

Comandos:

- Variables: `n`, `i`.
- Operadores: `**`, `+`.
- Funciones: `def`, `return`, `range()`, `print()`.

Ejercicio 37: Números primos con función

Enunciado:
Crea una función que devuelva todos los números primos hasta n.

Código:

```
def primos_hasta(n):
    primos = []
    for i in range(2, n+1):
        es_primo = True
        for j in range(2, i):
            if i % j == 0:
                es_primo = False
                break
        if es_primo:
            primos.append(i)
    return primos


print(primos_hasta(20))
```

Explicación:

- Función recorre cada número y verifica si es primo.
- `.append()` agrega el número a la lista de primos.

Comandos:

- Variables: `primos`, `i`, `j`, `es_primo`.
- Operadores: `%`, `==`.
- Funciones: `def`, `append()`, `range()`, `print()`.
- Estructuras: `for`, `if`, `break`.

Ejercicio 38: Contar palabras repetidas

Enunciado:
Cuenta cuántas veces se repite cada palabra en una frase.

Código:

```
frase = input("Ingrese una frase: ").lower().split()
contador = {}

for palabra in frase:
    contador[palabra] = contador.get(palabra, 0) + 1

print(contador)
```

Explicación:

- `.split()` separa la frase en palabras.
- `.get(palabra, 0)` devuelve el conteo actual o 0 si no existe.

Comandos:

- Variables: `frase`, `contador`, `palabra`.
- Operadores: `+`.
- Funciones: `input()`, `lower()`, `split()`, `get()`, `print()`.
- Estructuras: `for`.

Ejercicio 39: Ordenar diccionario por valor

Enunciado:
Ordena un diccionario por sus valores de menor a mayor.

Código:

```
edades = {"Ana": 20, "Luis": 25, "Marta": 30}
ordenado = dict(sorted(edades.items(), key=lambda x: x[1]))
print(ordenado)
```

Explicación:

- `items()` devuelve tuplas `(clave, valor)`.
- `sorted()` ordena según el segundo elemento de la tupla.

Comandos:

- Variables: `edades`, `ordenado`.
- Funciones: `items()`, `sorted()`, `dict()`, `print()`.
- Operadores: `:` (lambda).

Ejercicio 40: Contar números negativos y positivos

Enunciado:
Cuenta cuántos números negativos y positivos hay en una lista.

Código:

```
numeros = [-2, 3, -1, 5, 0]
positivos = sum(1 for n in numeros if n > 0)
negativos = sum(1 for n in numeros if n < 0)
print("Positivos:", positivos)
print("Negativos:", negativos)
```

Explicación:

- List comprehension dentro de `sum()` cuenta los elementos que cumplen la condición.

Comandos:

- Variables: `numeros`, `positivos`, `negativos`, `n`.
- Operadores: `>`, `<`.
- Funciones: `sum()`, `print()`.
- Estructuras: `if` dentro de la comprensión.

Ejercicio 41: Guardar lista en archivo

Enunciado:
Guarda los elementos de una lista en un archivo, uno por línea.

Código:

```
lista = ["manzana", "pera", "naranja"]


with open("frutas.txt", "w") as archivo:
    for item in lista:
        archivo.write(item + "\n")


print("Lista guardada en frutas.txt")
```

Explicación:

- Recorremos la lista con `for` y escribimos cada elemento en el archivo añadiendo un salto de línea `\n`.

Comandos:

- Variables: `lista`, `archivo`, `item`.
- Operadores: `+`.
- Funciones: `open()`, `write()`, `print()`.
- Estructuras: `with`, `for`.

Ejercicio 42: Leer archivo línea por línea

Enunciado:
Lee un archivo y muestra cada línea en pantalla.

Código:

```
with open("frutas.txt", "r") as archivo:
    for linea in archivo:
        print(linea.strip())
```

Explicación:

- `.strip()` elimina saltos de línea y espacios al inicio y final.
- `for linea in archivo` recorre cada línea del archivo.

Comandos:

- Variables: `archivo`, `linea`.
- Funciones: `open()`, `strip()`, `print()`.
- Estructuras: `with`, `for`.

Ejercicio 43: Validar entrada de número

Enunciado:
Pide un número al usuario hasta que ingrese uno válido.

Código:

```
while True:
    try:
        num = int(input("Ingrese un número: "))
        break
    except ValueError:
        print("¡Eso no es un número!")

print("Número ingresado:", num)
```

Explicación:

- `while True` crea un bucle infinito que se rompe con `break` al ingresar un número válido.

Comandos:

- Variables: `num`.
- Operadores: ninguno.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `while`, `try` / `except`, `break`.

Ejercicio 44: Lista de diccionarios

Enunciado:
Crea una lista de diccionarios con nombres y edades, y muestra solo los mayores de 18.

Código:

```
personas = [{"nombre":"Ana", "edad":20}, {"nombre":"Luis", "edad":15},
{"nombre":"Marta", "edad":30}]


mayores = [p["nombre"] for p in personas if p["edad"] > 18]

print("Mayores de 18:", mayores)
```

Explicación:

- List comprehension filtra los diccionarios según la edad.

Comandos:

- Variables: `personas`, `mayores`, `p`.
- Operadores: `>`.
- Funciones: `print()`.
- Estructuras: `for`, `if` dentro de list comprehension.

Ejercicio 45: Sumar valores de diccionario

Enunciado:
Suma los valores de un diccionario de productos con su cantidad vendida.

Código:

```
ventas = {"manzanas": 10, "peras": 5, "naranjas": 8}
total = sum(ventas.values())
print("Total de ventas:", total)
```

Explicación:

- `.values()` obtiene los valores, y `sum()` los suma.

Comandos:

- Variables: `ventas`, `total`.
- Funciones: `values()`, `sum()`, `print()`.

Ejercicio 46: Crear diccionario de frecuencias

Enunciado:
Cuenta la frecuencia de cada número en una lista.

Código:

```
numeros = [1,2,2,3,3,3,4]
frecuencia = {}

for n in numeros:
    frecuencia[n] = frecuencia.get(n,0) + 1

print(frecuencia)
```

Explicación:

- `.get(n,0)` devuelve el valor actual o 0 si no existe la clave.

Comandos:

- Variables: `numeros`, `frecuencia`, `n`.
- Operadores: `+`.
- Funciones: `get()`, `print()`.
- Estructuras: `for`.

Ejercicio 47: Función para buscar en lista

Enunciado:
Crea una función que busque un elemento en una lista y devuelva su índice o `-1` si no se encuentra.

Código:

```
def buscar(lista, valor):
    if valor in lista:
        return lista.index(valor)
    else:
        return -1


print(buscar([10,20,30], 20))
print(buscar([10,20,30], 40))
```

Explicación:

- `.index()` devuelve la posición de un elemento en la lista.

Comandos:

- Variables: `lista`, `valor`.
- Operadores: `in`.
- Funciones: `def`, `return`, `index()`, `print()`.
- Estructuras: `if` / `else`.

Ejercicio 48: Merge de diccionarios

Enunciado:
Combina dos diccionarios en uno solo.

Código:

```
d1 = {"a":1, "b":2}
d2 = {"b":3, "c":4}
d1.update(d2)
print(d1)
```

Explicación:

- `.update()` agrega las claves de `d2` a `d1`, reemplazando valores de claves repetidas.

Comandos:

- Variables: `d1`, `d2`.
- Funciones: `update()`, `print()`.

Ejercicio 49: Eliminar duplicados de lista

Enunciado:
Crea una lista sin elementos duplicados a partir de otra lista.

Código:

```
lista = [1,2,2,3,4,4,5]

lista_sin_duplicados = list(set(lista))

print(lista_sin_duplicados)
```

Explicación:

- `set()` elimina duplicados.
- `list()` convierte el conjunto de nuevo a lista.

Comandos:

- Variables: `lista`, `lista_sin_duplicados`.
- Funciones: `set()`, `list()`, `print()`.

Ejercicio 50: Promedio de edades desde archivo

Enunciado:
Lee edades desde un archivo y calcula el promedio.

Código:

```
with open("edades.txt", "r") as archivo:
    edades = [int(linea.strip()) for linea in archivo]

promedio = sum(edades)/len(edades)
print("Promedio de edades:", promedio)
```

Explicación:

- Leemos cada línea del archivo, convertimos a entero y calculamos el promedio con `sum()` y `len()`.

Comandos:

- Variables: `archivo`, `edades`, `promedio`, `linea`.
- Operadores: `/`.
- Funciones: `open()`, `strip()`, `int()`, `sum()`, `len()`, `print()`.
- Estructuras: `with`, list comprehension.

MANUAL DETALLADO DE EJERCICIOS DE PYTHON: ESTRUCTURAS CONDICIONALES

EJERCICIOS 1 AL 10

Ejercicio 1: Número positivo o negativo

Enunciado: Determina si un número ingresado es positivo o negativo. **Código:**

```
num = float(input("Ingrese un número: "))
if num > 0:     print("El número es positivo") else:     print("El número es negativo o cero")
```

Explicación Minuciosa: La estructura `if` evalúa si la variable `num` es mayor que 0. Si esta condición se cumple, se imprime que el número es positivo. El bloque `else` se ejecuta automáticamente si la condición principal no se cumple, indicando que el número es negativo o cero. **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** `>` (Mayor que).
- **Funciones:** `input()` (Captura de entrada), `float()` (Conversión a punto flotante), `print()` (Mostrar salida).
- **Estructuras:** `if`, `else`.

Ejercicio 2: Par o impar

Enunciado: Determina si un número ingresado es par o impar. **Explicación Minuciosa:** Se utiliza el **operador módulo** (`%`) para obtener el residuo de la división del número entre 2. Si el residuo es 0 (`num % 2 == 0`), el número es par. Si la condición no se cumple, el número es impar. **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** `%` (Módulo/Residuo), `==` (Igualdad).
- **Funciones:** `input()`, `int()` (Conversión a entero), `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 3: Mayor de dos números

Enunciado: Muestra cuál de dos números es mayor. **Explicación Minuciosa:** El código compara el primer número (`a`) y el segundo número (`b`) utilizando el operador `>`. El bloque correspondiente a la condición verdadera se ejecuta, imprimiendo el mayor. **Controles y Comandos:**

- **Variables:** `a`, `b`.
- **Operadores:** `>`.
- **Funciones:** `input()`, `float()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 4: Mayor de tres números

Enunciado: Encuentra el mayor de tres números. **Explicación Minuciosa:** Se utiliza el operador lógico `and` para combinar condiciones, verificando si `a` es mayor o igual (`>=`) tanto a `b` como a `c`. Si esa condición no se cumple, `elif` (estructura condicional secundaria) permite evaluar si `b` es mayor o igual a `a` y a `c`. Finalmente, `else` captura el caso restante, donde `c` es el mayor. **Controles y Comandos:**

- **Variables:** `a`, `b`, `c`.
- **Operadores:** `>=` (Mayor o igual), `and` (Conjunción lógica).
- **Funciones:** `input()`, `float()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 5: Aprobado o suspendido

Enunciado: Determina si un estudiante aprueba un examen (nota ≥ 5). **Explicación Minuciosa:** El código compara la variable `nota` con el valor 5. Si la nota es mayor o igual a 5 (`nota >= 5`), el estudiante es "Aprobado"; en caso contrario, es "Suspendido". **Controles y Comandos:**

- **Variables:** `nota`.
- **Operadores:** `>=`.
- **Funciones:** `input()`, `float()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 6: Clasificación de edad

Enunciado: Clasifica la edad en niño (<12), adolescente (12-17) o adulto (≥18). **Explicación Minuciosa:** El `if` inicial verifica si la edad es menor de 12. El `elif` se usa para el rango intermedio de adolescentes, evaluando si la edad es menor o igual a 17. El bloque `else` captura el resto de los casos, que son edades de adulto. **Controles y Comandos:**

- **Variables:** `edad`.
- **Operadores:** `<`, `<=`, `>=`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 7: Número múltiplo de 3

Enunciado: Determina si un número es múltiplo de 3. **Explicación Minuciosa:** Se utiliza el operador módulo (`%`) para calcular el residuo de la división del número entre 3. Si el residuo es 0, significa que el número es múltiplo de 3. **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** `%`, `==`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 8: Verificar contraseña

Enunciado: Comprueba si la contraseña ingresada es correcta. **Explicación Minuciosa:** Se utiliza el operador de igualdad (`==`) para comparar la cadena ingresada (`contraseña`) con la cadena

predefinida ("python123"). Si las cadenas coinciden, se concede el acceso; en caso contrario, se deniega. **Controles y Comandos:**

- **Variables:** `contraseña`.
- **Operadores:** `==`.
- **Funciones:** `input()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 9: Número entre 1 y 10

Enunciado: Verifica si un número está entre 1 y 10. **Explicación Minuciosa:** La **comparación encadenada** `1 <= num <= 10` verifica el rango de forma simultánea en una sola expresión. Si el número cumple la condición de estar en el rango, se imprime la afirmación. **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** `<=` (Menor o igual).
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 10: Número divisible por 2 y 5

Enunciado: Verifica si un número es divisible por 2 y 5 al mismo tiempo. **Explicación Minuciosa:** Se usa el operador `and` para requerir que **ambas condiciones** de divisibilidad sean verdaderas: que el residuo al dividir por 2 sea 0, Y que el residuo al dividir por 5 sea 0. **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** `%`, `==`, `and`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `else`.

EJERCICIOS 11 AL 20

Ejercicio 11: Número positivo, negativo o cero

Enunciado: Determina si un número ingresado es positivo, negativo o cero. **Explicación Minuciosa:** El `if` verifica si es positivo (`> 0`). El `elif` evalúa la segunda condición si la primera falla, verificando si es negativo (`< 0`). El `else` captura el resto de los casos, que en este contexto significa que el número es cero. **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** `>`, `<`.
- **Funciones:** `input()`, `float()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 12: Número divisible por 2 o 3

Enunciado: Verifica si un número es divisible por 2 o por 3. **Explicación Minuciosa:** El operador lógico `or` se utiliza para permitir que **cualquiera** de las condiciones de divisibilidad sea verdadera (divisible por 2 O divisible por 3). **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** `%`, `==`, `or` (Disyunción lógica).
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 13: Comparar tres números

Enunciado: Indica cuál de tres números es el mayor y si hay empate. **Explicación Minuciosa:** Se usan múltiples `if`/`elif` junto con el operador `and` para combinar las condiciones de comparación y determinar qué variable es estrictamente mayor que las otras dos. El bloque `else` está diseñado para capturar el caso en que no se cumple ninguna de las condiciones de estricta superioridad, indicando que hay un empate entre los números. **Controles y Comandos:**

- **Variables:** `a`, `b`, `c`.
- **Operadores:** `>`, `and`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 14: Clasificación de edades detallada

Enunciado: Clasifica la edad en niño (<12), adolescente (12-17), joven adulto (18-25) o adulto (>25). **Explicación Minuciosa:** Se utilizan múltiples `elif` para evaluar rangos de edad distintos y consecutivos. El `else` final captura cualquier edad que sea mayor que el límite superior del último `elif` (en este caso, mayor de 25). **Controles y Comandos:**

- **Variables:** `edad`.
- **Operadores:** `<`, `<=`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 15: Evaluar nota con calificación

Enunciado: Convierte una nota (0-10) en calificación: A ≥9, B ≥7, C ≥5, F <5. **Explicación Minuciosa:** Las condiciones se evalúan en rangos descendentes. Se utiliza `elif` para evaluar múltiples niveles de calificación, comprobando si la nota es mayor o igual al umbral de cada calificación (A, B, C). Si ninguna de las condiciones anteriores se cumple, el `else` asigna la calificación F (menor a 5). **Controles y Comandos:**

- **Variables:** `nota`.
- **Operadores:** `>=`, `<`.
- **Funciones:** `input()`, `float()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 16: Número dentro o fuera de rango

Enunciado: Verifica si un número está dentro del rango 10-50 inclusive. **Explicación Minuciosa:** La expresión `10 <= num <= 50` es una **comparación encadenada** que evalúa si el número está en el rango especificado (incluyendo los límites 10 y 50) en una sola línea. **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** `<=`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 17: Contraseña y usuario

Enunciado: Verifica si el usuario y la contraseña son correctos. **Explicación Minuciosa:** El operador `and` es fundamental, ya que requiere que **ambas condiciones** (`usuario == "admin"` Y `contraseña == "1234"`) sean verdaderas simultáneamente para conceder el acceso. **Controles y Comandos:**

- **Variables:** `usuario`, `contraseña`.
- **Operadores:** `==`, `and`.
- **Funciones:** `input()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 18: Número divisible por 4 y no por 8

Enunciado: Verifica si un número es divisible por 4 pero no por 8. **Explicación Minuciosa:** Se combinan dos condiciones mediante `and`. La primera condición verifica la divisibilidad por 4 (`num % 4 == 0`), y la segunda condición utiliza el operador `!=` (diferente de) para asegurar que el número *no* sea divisible por 8 (`num % 8 != 0`). **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** `%`, `==`, `!=` (Diferente de), `and`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 19: Determinar estación del año

Enunciado: Muestra la estación del año según el mes ingresado (1-12). **Explicación Minuciosa:** Se utiliza el operador `in` para verificar si el valor de la variable `mes` está contenido dentro de una de las listas de meses predefinidas. Múltiples `elif` permiten evaluar cada uno de los rangos de meses correspondientes a las estaciones. **Controles y Comandos:**

- **Variables:** `mes`.
- **Operadores:** `in`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 20: Aprobado, examen final o suspendido

Enunciado: Nota ≥7 aprobado, 5-6 examen final, <5 suspendido. **Explicación Minuciosa:** El `if` inicial maneja la condición de aprobado (≥ 7). El `elif` posterior, debido al flujo de control, captura automáticamente el rango intermedio (notas entre 5 y 6.9, ya que las notas ≥ 7 fueron descartadas). El `else` final cubre cualquier nota menor a 5. **Controles y Comandos:**

- **Variables:** `nota`.
- **Operadores:** `>=`.
- **Funciones:** `input()`, `float()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

EJERCICIOS 21 AL 30

Ejercicio 21: Número entre 10-20 o 30-40

Enunciado: Verifica si un número está en los rangos 10-20 o 30-40. **Explicación Minuciosa:** Se definen dos condiciones de rango independientes usando comparaciones encadenadas (`10 <= num <= 20` y `30 <= num <= 40`). El operador `or` permite que el número sea considerado "en el rango" si **cualquiera** de estas dos condiciones es verdadera. Los paréntesis se usan para aclarar la precedencia de las operaciones. **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** `<=`, `or`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 22: Número no divisible por 2 y 5

Enunciado: Verifica si un número no es divisible ni por 2 ni por 5. **Explicación Minuciosa:** Se verifica primero si el número es divisible por 2 O por 5 (usando `or`). Luego, el operador `not` se utiliza para **invertir** el resultado de esta condición combinada. Solo si la condición combinada es falsa (es decir, no es divisible ni por 2 ni por 5), el `if` se ejecuta. **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** `%`, `==`, `or`, `not` (Negación lógica).
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 23: Edad válida para conducir

Enunciado: Verifica si alguien tiene edad válida (18-65) para conducir. **Explicación Minuciosa:** Se utiliza la **comparación encadenada** `18 <= edad <= 65` para validar el rango de edad en una única expresión. **Controles y Comandos:**

- **Variables:** `edad`.
- **Operadores:** `<=`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 24: Clasificar número como pequeño, mediano o grande

Enunciado: Número <10 pequeño, 10-50 mediano, >50 grande. **Explicación Minuciosa:** El `if` verifica el rango "pequeño" (< 10). El `elif` utiliza rangos encadenados (`10 <= num <= 50`) para definir el caso "mediano". El `else` captura todos los números restantes, que son mayores de 50 ("grande"). **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** `<`, `<=`, `>=`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 25: Determinar tipo de triángulo

Enunciado: Determina si un triángulo es equilátero, isósceles o escaleno según sus lados.
Explicación Minuciosa:

1. **Equilátero:** Se usa el operador `==` encadenado (`a == b == c`) para verificar que los tres lados son iguales.
2. **Isósceles:** Si no es equilátero, el `elif` verifica si al menos un par de lados es igual usando el operador `or` (`a == b or b == c or a == c`).
3. **Escaleno:** El `else` captura el caso donde ninguno de los pares es igual (escaleno).
   Controles y Comandos:

- **Variables:** `a`, `b`, `c`.
- **Operadores:** `==`, `or`.
- **Funciones:** `input()`, `float()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 26: Verificar vocal

Enunciado: Determina si un carácter ingresado es una vocal. **Explicación Minuciosa:** La función `.lower()` se aplica a la entrada para convertir la letra a minúscula, asegurando una comparación uniforme. El operador `in` comprueba eficientemente si el carácter está contenido dentro de la cadena `"aeiou"`. **Controles y Comandos:**

- **Variables:** `letra`.
- **Operadores:** `in`.
- **Funciones:** `input()`, `lower()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 27: Determinar trimestre del año

Enunciado: Muestra a qué trimestre pertenece un mes (1-12). **Explicación Minuciosa:** Se utiliza una secuencia de `elif`s, donde cada uno evalúa un rango de tres meses utilizando comparaciones encadenadas (`1 <= mes <= 3`, etc.). Si el número de mes no cae en ninguno de los trimestres (1-12), el `else` indica un mes inválido. **Controles y Comandos:**

- **Variables:** `mes`.
- **Operadores:** `<=`.

- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 28: Número primo (condición simple)

Enunciado: Verifica si un número pequeño es primo (2-7). **Explicación Minuciosa:** Este ejercicio usa una condición simple basada en una lista predefinida de primos dentro del rango (2, 3, 5, 7). El operador `in` comprueba si el número ingresado se encuentra en esa lista de primos conocidos. **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** `in`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 29: Evaluar nota con recuperación

Enunciado: Nota ≥7 aprobado, 5-6 examen de recuperación, <5 suspendido. **Explicación Minuciosa:** El `elif` verifica las notas intermedias usando el rango encadenado `5 <= nota < 7`. Esto garantiza que solo las notas que son 5 o más, pero estrictamente menores que 7, caigan en la categoría de "Recuperación". **Controles y Comandos:**

- **Variables:** `nota`.
- **Operadores:** >=, <=, <.
- **Funciones:** `input()`, `float()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 30: Número positivo, negativo o múltiplo de 5

Enunciado: Determina si un número es positivo, negativo o múltiplo de 5. **Explicación Minuciosa:** Se utilizan múltiples `elif` para evaluar condiciones que pueden ser mutuamente excluyentes (por ejemplo, si es positivo o si es negativo). El tercer `elif` comprueba si es múltiplo de 5, usando el operador módulo (`num % 5 == 0`). El `else` final captura casos como el cero o números que no cumplen las condiciones previas. **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** >, <, %, ==.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

EJERCICIOS 31 AL 40

Ejercicio 31: Determinar signo de tres números

Enunciado: Indica si los tres números ingresados son positivos, negativos o mixtos. **Explicación Minuciosa:** Se emplea el operador `and` para combinar varias condiciones y asegurar que se cumplan simultáneamente:

1. Verificación de que todos son positivos (`a > 0 and b > 0 and c > 0`).

2. Verificación de que todos son negativos (`a < 0 and b < 0 and c < 0`). Si ninguna de las dos condiciones anteriores es cierta, el `else` indica que existe una mezcla de signos (positivos y negativos, o ceros). **Controles y Comandos:**

- **Variables:** `a`, `b`, `c`.
- **Operadores:** `>`, `<`, `and`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 32: Clasificación de notas avanzada

Enunciado: Convierte una nota en letras A-F considerando rangos específicos (A ≥ 9.5, B 8-9.4, C 6.5-7.9, D 5-6.4, F <5). **Explicación Minuciosa:** Se utilizan rangos encadenados con precisión decimal (`>=` y `<`) en los `elif` para definir los límites de cada calificación. Por ejemplo, para la calificación B, se verifica `8 <= nota < 9.5`. **Controles y Comandos:**

- **Variables:** `nota`.
- **Operadores:** `>=`, `<=`, `<`.
- **Funciones:** `input()`, `float()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 33: Verificar adulto con nacionalidad

Enunciado: Comprueba si una persona es mayor de edad (≥18) y tiene nacionalidad "española". **Explicación Minuciosa:** El operador `and` requiere que ambas condiciones se cumplan: la `edad >= 18` Y la `nacionalidad == "española"`. Se usa la función `.lower()` para normalizar la entrada de nacionalidad a minúsculas antes de la comparación. **Controles y Comandos:**

- **Variables:** `edad`, `nacionalidad`.
- **Operadores:** `>=`, `==`, `and`.
- **Funciones:** `input()`, `int()`, `lower()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 34: Evaluar temperatura

Enunciado: Clasifica la temperatura: fría (<15), templada (15-25), caliente (>25). **Explicación Minuciosa:** El `elif` utiliza la comparación encadenada `15 <= temp <= 25` para definir el rango intermedio "Templada". El `else` final abarca todas las temperaturas mayores a 25 ("Caliente"). **Controles y Comandos:**

- **Variables:** `temp`.
- **Operadores:** `<`, `<=`, `>=`.
- **Funciones:** `input()`, `float()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 35: Número divisible por 2, 3 o ambos

Enunciado: Verifica si un número es divisible por 2, 3, o ambos. **Explicación Minuciosa:** Se evalúa primero la condición más restrictiva: divisibilidad por 2 **y** 3 (usando `and`). Los `elif`

posteriores evalúan si el número es divisible **solo** por 2 o **solo** por 3, aprovechando que los casos de divisibilidad dual ya fueron manejados. **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** `%`, `==`, `and`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 36: Determinar triángulo válido

Enunciado: Verifica si tres lados pueden formar un triángulo (suma de dos lados > tercer lado). **Explicación Minuciosa:** Se utiliza el operador `and` para combinar las tres condiciones necesarias de desigualdad triangular: `a + b > c`, `a + c > b`, y `b + c > a`. Si las tres desigualdades se cumplen simultáneamente, el triángulo es válido. **Controles y Comandos:**

- **Variables:** `a`, `b`, `c`.
- **Operadores:** `+` (Suma), `>` (Mayor que), `and`.
- **Funciones:** `input()`, `float()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 37: Determinar si año es bisiesto

Enunciado: Verifica si un año es bisiesto: divisible por 4 y no por 100, salvo que sea divisible por 400. **Explicación Minuciosa:** La regla de bisiesto requiere una combinación compleja de operadores `and` y `or`. La condición general es: (divisible por 4 Y no divisible por 100) O (divisible por 400). Se usan paréntesis para asegurar la correcta precedencia lógica de las subcondiciones. **Controles y Comandos:**

- **Variables:** `año`.
- **Operadores:** `%`, `==`, `!=`, `and`, `or`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 38: Aceptar o rechazar entrada

Enunciado: Acepta solo números entre 1 y 100, múltiplo de 5. **Explicación Minuciosa:** Se combinan dos requisitos usando `and`:

1. Rango: `1 <= num <= 100`.
2. Divisibilidad: `num % 5 == 0`. Ambas condiciones deben ser verdaderas para que el número se considere válido. **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** `<=`, `>=`, `%`, `==`, `and`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicio 39: Clasificación de altura

Enunciado: Clasifica altura en baja (<1.60 m), media (1.60-1.80 m), alta (>1.80 m). **Explicación Minuciosa:** El `elif` utiliza rangos encadenados (`1.60 <= altura <= 1.80`) para gestionar la clasificación "Media". Los `if/elif/else` permiten clasificar la altura en estas tres categorías discretas. **Controles y Comandos:**

- **Variables:** `altura`.
- **Operadores:** <, <=, >=.
- **Funciones:** `input()`, `float()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 40: Edad para carnet de conducir especial

Enunciado: Verifica si alguien puede obtener un carnet especial (18-70) y no tiene multas (multas = 0). **Explicación Minuciosa:** Se usa `and` para combinar el requisito de rango de edad (`18 <= edad <= 70`) con la condición de que el número de multas sea cero (`multas == 0`). Ambas condiciones deben cumplirse para la obtención del carnet. **Controles y Comandos:**

- **Variables:** `edad`, `multas`.
- **Operadores:** <=, >=, ==, `and`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `else`.

EJERCICIOS 41 AL 50

Ejercicio 41: Determinar el trimestre académico

Enunciado: Indica el trimestre académico según el mes (1-12). **Explicación Minuciosa:** Similar al Ejercicio 27, se utiliza una secuencia de estructuras `elif`, donde cada bloque condicional evalúa un rango de tres meses consecutivos (por ejemplo, `1 <= mes <= 3` para el 1er trimestre). **Controles y Comandos:**

- **Variables:** `mes`.
- **Operadores:** <=, >=.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 42: Determinar día de la semana

Enunciado: Recibe un número (1-7) y devuelve el día de la semana. **Explicación Minuciosa:** Se utiliza una larga cadena de `elif`s para evaluar cada opción de número individualmente, usando el operador de igualdad (==) para mapear cada número (1 a 7) a su respectivo día de la semana. El `else` maneja entradas fuera del rango 1-7. **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** ==.
- **Funciones:** `input()`, `int()`, `print()`.

- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 43: Aceptar o rechazar acceso (Anidado)

Enunciado: Valida usuario y contraseña. Ambos deben ser correctos. **Explicación Minuciosa:** Se utiliza una **estructura condicional anidada**. El `if` principal verifica el `usuario`. Solo si el usuario es correcto, el flujo entra al `if` anidado para verificar la `contraseña`. Esto permite diferenciar si el error fue por el usuario (primer `else`) o por la contraseña (segundo `else`). **Controles y Comandos:**

- **Variables:** `usuario`, `contraseña`.
- **Operadores:** ==.
- **Funciones:** `input()`, `print()`.
- **Estructuras:** `if` anidado, `else`.

Ejercicio 44: Clasificación de edades múltiple

Enunciado: Clasifica edades: niño (<12), adolescente (12-17), joven (18-30), adulto (31-60), mayor (>60). **Explicación Minuciosa:** Se usan condiciones encadenadas (`if/elif/else`) para definir múltiples rangos de edad. Cada `elif` subsiguiente se basa en que las condiciones anteriores (rangos inferiores) ya han sido descartadas. **Controles y Comandos:**

- **Variables:** `edad`.
- **Operadores:** <, <=.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 45: Clasificación de notas con recuperación

Enunciado: Clasificación: ≥9: Excelente, 7-8.9: Bueno, 5-6.9: Regular, <5: Insuficiente. **Explicación Minuciosa:** Para los rangos intermedios, se utilizan combinaciones de operadores <= y < (e.g., `7 <= nota < 9` para "Bueno"). Esto define con precisión los límites de cada categoría de nota. **Controles y Comandos:**

- **Variables:** `nota`.
- **Operadores:** >=, <=, <.
- **Funciones:** `input()`, `float()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 46: Número divisible por 2, 3 y 5

Enunciado: Verifica divisibilidad múltiple y específica. **Explicación Minuciosa:** Se evalúan las condiciones de divisibilidad de mayor a menor restricción. El primer `if` comprueba la divisibilidad simultánea por 2, 3, y 5 usando múltiples `and`. El siguiente `elif` comprueba el caso de ser divisible solo por 2 y 3, y así sucesivamente, para aislar la especificidad de la divisibilidad. **Controles y Comandos:**

- **Variables:** `num`.
- **Operadores:** %, ==, `and`.
- **Funciones:** `input()`, `int()`, `print()`.

- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 47: Validación de contraseña fuerte

Enunciado: Contraseña válida si tiene ≥8 caracteres y contiene al menos un número. **Explicación Minuciosa:** Se emplea una **estructura anidada** para validar las reglas.

1. El `if` exterior usa `len()` para obtener la longitud de la cadena y verifica si es `>= 8`.
2. Si la longitud es suficiente, el `if` interior usa la función `any()` combinada con la función `.isdigit()` (aplicada a cada carácter) para verificar si la contraseña contiene al menos un dígito. **Controles y Comandos:**

- **Variables:** `contraseña`, `c`.
- **Operadores:** `>=`.
- **Funciones:** `input()`, `len()` (Longitud), `any()` (Verifica si algún elemento es verdadero), `isdigit()` (Verifica si es dígito), `print()`.
- **Estructuras:** `if` anidado, `else`.

Ejercicio 48: Mayor de tres números con igualdad

Enunciado: Determina el mayor de tres números y si hay empate. **Explicación Minuciosa:** Se comprueba primero la igualdad total (`a == b == c`). Si no hay igualdad total, se usan `elif` para determinar el mayor, permitiendo la igualdad (`>=`) para cubrir los empates parciales (e.g., `a` es mayor o igual a `b` Y mayor o igual a `c`). El `else` final cubre el caso en que `c` es el mayor (incluyendo empates en los que `c` está involucrado y no se cubrieron antes). **Controles y Comandos:**

- **Variables:** `a`, `b`, `c`.
- **Operadores:** `==`, `>=`, `and`.
- **Funciones:** `input()`, `int()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 49: Clasificación de temperatura

Enunciado: Clasifica la temperatura en Celsius en cinco categorías: Congelante (<0), Fría (0-15), Templada (16-25), Caliente (26-35), Muy caliente (>35). **Explicación Minuciosa:** Este ejercicio utiliza múltiples `elif` con rangos encadenados para clasificar la temperatura con alta granularidad, asegurando que cada rango de temperatura sea exclusivo y cubra el espectro completo (e.g., `0 <= temp <= 15`, `16 <= temp <= 25`, etc.). **Controles y Comandos:**

- **Variables:** `temp`.
- **Operadores:** `<`, `<=`, `>=`.
- **Funciones:** `input()`, `float()`, `print()`.
- **Estructuras:** `if`, `elif`, `else`.

Ejercicio 50: Acceso según edad y membresía

Enunciado: Acceso permitido si edad ≥18 y tiene membresía activa. **Explicación Minuciosa:** Se utiliza el operador `and` para imponer la doble condición de acceso: la `edad >= 18` Y la

`membresia` debe ser igual a "sí". Se aplica la función `.lower()` a la entrada de la membresía para estandarizar la comparación, aceptando entradas como "SÍ" o "sí". **Controles y Comandos:**

- **Variables:** `edad`, `membresia`.
- **Operadores:** `>=`, `==`, `and`.
- **Funciones:** `input()`, `int()`, `lower()`, `print()`.
- **Estructuras:** `if`, `else`.

Ejercicios de Python, estructuras condicionales

Ejercicio 1: Número positivo o negativo

Enunciado:
Determina si un número ingresado es positivo o negativo.

Código:

```
num = float(input("Ingrese un número: "))

if num > 0:
    print("El número es positivo")
else:
    print("El número es negativo o cero")
```

Explicación:

- `if` evalúa si el número es mayor que 0.
- `else` se ejecuta si la condición no se cumple.

Comandos:

- Variables: `num`.
- Operadores: `>`.
- Funciones: `input()`, `float()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 2: Par o impar

Enunciado:
Determina si un número ingresado es par o impar.

Código:

```
num = int(input("Ingrese un número: "))

if num % 2 == 0:
    print("El número es par")
else:
    print("El número es impar")
```

Explicación:

- % devuelve el residuo. Si es 0, el número es par.

Comandos:

- Variables: `num`.
- Operadores: `%`, `==`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 3: Mayor de dos números

Enunciado:
Muestra cuál de dos números es mayor.

Código:

```
a = float(input("Ingrese el primer número: "))
b = float(input("Ingrese el segundo número: "))

if a > b:
    print("El mayor es", a)
else:
    print("El mayor es", b)
```

Explicación:

- Compara `a` y `b` con `>`.
- Ejecuta el bloque correspondiente.

Comandos:

- Variables: `a`, `b`.
- Operadores: `>`.
- Funciones: `input()`, `float()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 4: Mayor de tres números

Enunciado:
Encuentra el mayor de tres números.

Código:

```
a = float(input("Número 1: "))
b = float(input("Número 2: "))
c = float(input("Número 3: "))

if a >= b and a >= c:
    print("El mayor es", a)
elif b >= a and b >= c:
    print("El mayor es", b)
else:
    print("El mayor es", c)
```

Explicación:

- `and` combina condiciones.
- `elif` permite evaluar otra condición si la primera no se cumple.

Comandos:

- Variables: `a`, `b`, `c`.
- Operadores: `>=`, `and`.
- Funciones: `input()`, `float()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 5: Aprobado o suspendido

Enunciado:
Determina si un estudiante aprueba un examen (nota ≥ 5).

Código:

```
nota = float(input("Ingrese la nota: "))

if nota >= 5:
    print("Aprobado")
else:
    print("Suspendido")
```

Explicación:

- Compara la nota con 5.

Comandos:

- Variables: `nota`.
- Operadores: `>=`.
- Funciones: `input()`, `float()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 6: Clasificación de edad

Enunciado:
Clasifica la edad en niño (<12), adolescente (12-17) o adulto (≥18).

Código:

```
edad = int(input("Ingrese la edad: "))

if edad < 12:
    print("Niño")
elif edad <= 17:
    print("Adolescente")
else:
    print("Adulto")
```

Explicación:

- `elif` se usa para rango de edades intermedio.

Comandos:

- Variables: `edad`.
- Operadores: `<`, `<=`, `>=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 7: Número múltiplo de 3

Enunciado:
Determina si un número es múltiplo de 3.

Código:

```
num = int(input("Ingrese un número: "))

if num % 3 == 0:
    print("Es múltiplo de 3")
else:
    print("No es múltiplo de 3")
```

Explicación:

- % calcula residuo. Si es 0, es múltiplo.

Comandos:

- Variables: `num`.
- Operadores: `%`, `==`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 8: Verificar contraseña

Enunciado:
Comprueba si la contraseña ingresada es correcta.

Código:

```
contraseña = input("Ingrese la contraseña: ")

if contraseña == "python123":
    print("Acceso concedido")
else:
    print("Acceso denegado")
```

Explicación:

- `==` compara cadenas.

Comandos:

- Variables: `contraseña`.
- Operadores: `==`.
- Funciones: `input()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 9: Número entre 1 y 10

Enunciado:
Verifica si un número está entre 1 y 10.

Código:

```
num = int(input("Ingrese un número: "))

if 1 <= num <= 10:
    print("Está entre 1 y 10")
else:
    print("No está en el rango")
```

Explicación:

- `1 <= num <= 10` verifica rango simultáneamente.

Comandos:

- Variables: `num`.
- Operadores: `<=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 10: Número divisible por 2 y 5

Enunciado:
Verifica si un número es divisible por 2 y 5 al mismo tiempo.

Código:

```
num = int(input("Ingrese un número: "))

if num % 2 == 0 and num % 5 == 0:
    print("Divisible por 2 y 5")
else:
    print("No es divisible por 2 y 5")
```

Explicación:

- `and` requiere que ambas condiciones sean verdaderas.

Comandos:

- Variables: `num`.
- Operadores: `%`, `==`, `and`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 11: Número positivo, negativo o cero

Enunciado:
Determina si un número ingresado es positivo, negativo o cero.

Código:

```
num = float(input("Ingrese un número: "))


if num > 0:
    print("Positivo")
elif num < 0:
    print("Negativo")
else:
    print("Cero")
```

Explicación:

- `elif` permite evaluar la segunda condición si la primera no se cumple.
- `else` captura el resto de los casos.

Comandos:

- Variables: `num`.
- Operadores: `>`, `<`.
- Funciones: `input()`, `float()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 12: Número divisible por 2 o 3

Enunciado:
Verifica si un número es divisible por 2 o por 3.

Código:

```
num = int(input("Ingrese un número: "))


if num % 2 == 0 or num % 3 == 0:
    print("Divisible por 2 o 3")
else:
    print("No divisible por 2 ni 3")
```

Explicación:

- `or` permite que cualquiera de las condiciones sea verdadera.

Comandos:

- Variables: `num`.
- Operadores: `%`, `==`, `or`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 13: Comparar tres números

Enunciado:
Indica cuál de tres números es el mayor y si hay empate.

Código:

```
a = int(input("Número 1: "))
b = int(input("Número 2: "))
c = int(input("Número 3: "))

if a > b and a > c:
    print("El mayor es", a)
elif b > a and b > c:
    print("El mayor es", b)
elif c > a and c > b:
    print("El mayor es", c)
else:
    print("Hay empate entre los números")
```

Explicación:

- Se usan `and` para combinar condiciones de comparación.
- `else` captura el caso de empate.

Comandos:

- Variables: `a`, `b`, `c`.
- Operadores: `>`, `and`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 14: Clasificación de edades detallada

Enunciado:
Clasifica la edad en niño (<12), adolescente (12-17), joven adulto (18-25) o adulto (>25).

Código:

```
edad = int(input("Ingrese la edad: "))


if edad < 12:
    print("Niño")
elif edad <= 17:
    print("Adolescente")
elif edad <= 25:
    print("Joven adulto")
else:
    print("Adulto")
```

Explicación:

- Cada `elif` evalúa un rango distinto.
- `else` captura cualquier edad mayor.

Comandos:

- Variables: `edad`.
- Operadores: `<`, `<=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 15: Evaluar nota con calificación

Enunciado:
Convierte una nota (0-10) en calificación: A ≥9, B ≥7, C ≥5, F <5.

Código:

```
nota = float(input("Ingrese la nota: "))


if nota >= 9:
    print("Calificación: A")
elif nota >= 7:
    print("Calificación: B")
elif nota >= 5:
    print("Calificación: C")
else:
    print("Calificación: F")
```

Explicación:

- Se usan rangos descendentes.
- `elif` permite evaluar múltiples niveles.

Comandos:

- Variables: `nota`.
- Operadores: `>=`, `<`.
- Funciones: `input()`, `float()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 16: Número dentro o fuera de rango

Enunciado:
Verifica si un número está dentro del rango 10-50 inclusive.

Código:

```
num = int(input("Ingrese un número: "))


if 10 <= num <= 50:
    print("Dentro del rango")
else:
    print("Fuera del rango")
```

Explicación:

- Comparación encadenada `10 <= num <= 50` evalúa el rango en una sola expresión.

Comandos:

- Variables: `num`.
- Operadores: `<=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 17: Contraseña y usuario

Enunciado:
Verifica si el usuario y la contraseña son correctos.

Código:

```
usuario = input("Usuario: ")
contraseña = input("Contraseña: ")

if usuario == "admin" and contraseña == "1234":
    print("Acceso concedido")
else:
    print("Acceso denegado")
```

Explicación:

- `and` requiere que ambas condiciones sean verdaderas.

Comandos:

- Variables: `usuario`, `contraseña`.
- Operadores: `==`, `and`.
- Funciones: `input()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 18: Número divisible por 4 y no por 8

Enunciado:
Verifica si un número es divisible por 4 pero no por 8.

Código:

```
num = int(input("Ingrese un número: "))

if num % 4 == 0 and num % 8 != 0:
    print("Divisible por 4 y no por 8")
else:
    print("No cumple la condición")
```

Explicación:

- `!=` significa "diferente de".
- `and` combina ambas condiciones.

Comandos:

- Variables: `num`.
- Operadores: `%`, `==`, `!=`, `and`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 19: Determinar estación del año

Enunciado:
Muestra la estación del año según el mes ingresado (1-12).

Código:

```
mes = int(input("Ingrese el número del mes (1-12): "))


if mes in [12,1,2]:
    print("Invierno")
elif mes in [3,4,5]:
    print("Primavera")
elif mes in [6,7,8]:
    print("Verano")
elif mes in [9,10,11]:
    print("Otoño")
else:
    print("Mes inválido")
```

Explicación:

- `in` verifica si el valor está en la lista.
- `elif` permite múltiples rangos.

Comandos:

- Variables: `mes`.
- Operadores: `in`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 20: Aprobado, examen final o suspendido

Enunciado:
Nota ≥7 aprobado, 5-6 examen final, <5 suspendido.

Código:

```
nota = float(input("Ingrese la nota: "))


if nota >= 7:
    print("Aprobado")
elif nota >= 5:
    print("Examen final")
else:
    print("Suspendido")
```

Explicación:

- `elif` permite un rango intermedio.
- `else` cubre la nota menor a 5.

Comandos:

- Variables: `nota`.
- Operadores: `>=`.
- Funciones: `input()`, `float()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 21: Número entre 10-20 o 30-40

Enunciado:
Verifica si un número está en los rangos 10-20 o 30-40.

Código:

```
num = int(input("Ingrese un número: "))


if (10 <= num <= 20) or (30 <= num <= 40):
    print("El número está en el rango")
else:
    print("El número no está en el rango")
```

Explicación:

- `or` permite que cualquiera de los rangos sea válido.
- Paréntesis aclaran la precedencia.

Comandos:

- Variables: `num`.
- Operadores: `<=`, `or`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 22: Número no divisible por 2 y 5

Enunciado:
Verifica si un número **no** es divisible ni por 2 ni por 5.

Código:

```
num = int(input("Ingrese un número: "))

if not (num % 2 == 0 or num % 5 == 0):
    print("No es divisible ni por 2 ni por 5")
else:
    print("Es divisible por 2 o 5")
```

Explicación:

- `not` invierte la condición.
- `or` combina divisibilidad por 2 o 5.

Comandos:

- Variables: `num`.
- Operadores: `%`, `==`, `or`, `not`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 23: Edad válida para conducir

Enunciado:
Verifica si alguien tiene edad válida (18-65) para conducir.

Código:

```
edad = int(input("Ingrese la edad: "))


if 18 <= edad <= 65:
    print("Edad válida para conducir")
else:
    print("Edad no válida")
```

Explicación:

- Comparación encadenada `18 <= edad <= 65` verifica rango en una sola expresión.

Comandos:

- Variables: `edad`.
- Operadores: `<=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 24: Clasificar número como pequeño, mediano o grande

Enunciado:
Número <10 pequeño, 10-50 mediano, >50 grande.

Código:

```
num = int(input("Ingrese un número: "))


if num < 10:
    print("Pequeño")
elif 10 <= num <= 50:
    print("Mediano")
else:
    print("Grande")
```

Explicación:

- Se usan rangos encadenados y `elif` para el caso intermedio.

Comandos:

- Variables: `num`.
- Operadores: `<`, `<=`, `>=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 25: Determinar tipo de triángulo

Enunciado:
Determina si un triángulo es equilátero, isósceles o escaleno según sus lados.

Código:

```
a = float(input("Lado a: "))
b = float(input("Lado b: "))
c = float(input("Lado c: "))

if a == b == c:
    print("Equilátero")
elif a == b or b == c or a == c:
    print("Isósceles")
else:
    print("Escaleno")
```

Explicación:

- `==` encadenado verifica igualdad de tres lados.
- `or` evalúa igualdad de cualquier par.

Comandos:

- Variables: `a`, `b`, `c`.
- Operadores: `==`, `or`.
- Funciones: `input()`, `float()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 26: Verificar vocal

Enunciado:
Determina si un carácter ingresado es una vocal.

Código:

```
letra = input("Ingrese una letra: ").lower()


if letra in "aeiou":
    print("Es una vocal")
else:
    print("No es una vocal")
```

Explicación:

- `in` comprueba si el carácter está en la cadena de vocales.
- `.lower()` convierte la letra a minúscula.

Comandos:

- Variables: `letra`.
- Operadores: `in`.
- Funciones: `input()`, `lower()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 27: Determinar trimestre del año

Enunciado:
Muestra a qué trimestre pertenece un mes (1-12).

Código:

```
mes = int(input("Número de mes: "))


if 1 <= mes <= 3:
    print("Primer trimestre")
elif 4 <= mes <= 6:
    print("Segundo trimestre")
elif 7 <= mes <= 9:
    print("Tercer trimestre")
elif 10 <= mes <= 12:
    print("Cuarto trimestre")
else:
    print("Mes inválido")
```

Explicación:

- Cada `elif` evalúa un rango de tres meses.

Comandos:

- Variables: `mes`.
- Operadores: `<=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 28: Número primo (condición simple)

Enunciado:
Verifica si un número pequeño es primo (2-7).

Código:

```
num = int(input("Ingrese un número del 2 al 7: "))


if num in [2,3,5,7]:
    print("Es primo")
else:
    print("No es primo")
```

Explicación:

- `in` comprueba si el número está en la lista de primos conocidos.

Comandos:

- Variables: `num`.
- Operadores: `in`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 29: Evaluar nota con recuperación

Enunciado:
Nota ≥7 aprobado, 5-6 examen de recuperación, <5 suspendido.

Código:

```
nota = float(input("Ingrese la nota: "))


if nota >= 7:
    print("Aprobado")
elif 5 <= nota < 7:
    print("Recuperación")
else:
    print("Suspendido")
```

Explicación:

- Rango `5 <= nota < 7` verifica notas intermedias.

Comandos:

- Variables: `nota`.
- Operadores: `>=`, `<=`, `<`.
- Funciones: `input()`, `float()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 30: Número positivo, negativo o múltiplo de 5

Enunciado:
Determina si un número es positivo, negativo o múltiplo de 5.

Código:

```
num = int(input("Ingrese un número: "))


if num > 0:
    print("Positivo")
elif num < 0:
    print("Negativo")
elif num % 5 == 0:
    print("Múltiplo de 5")
else:
    print("Cero o no cumple condiciones")
```

Explicación:

- `elif` permite evaluar múltiples condiciones exclusivas.

Comandos:

- Variables: `num`.
- Operadores: `>`, `<`, `%`, `==`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 31: Determinar signo de tres números

Enunciado:
Indica si los tres números ingresados son positivos, negativos o mixtos.

Código:

```
a = int(input("Número 1: "))
b = int(input("Número 2: "))
c = int(input("Número 3: "))

if a > 0 and b > 0 and c > 0:
    print("Todos son positivos")
elif a < 0 and b < 0 and c < 0:
    print("Todos son negativos")
else:
    print("Mezcla de positivos y negativos")
```

Explicación:

- `and` combina varias condiciones que deben cumplirse simultáneamente.

Comandos:

- Variables: `a`, `b`, `c`.
- Operadores: `>`, `<`, `and`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 32: Clasificación de notas avanzada

Enunciado:
Convierte una nota en letras A-F considerando ±:

- A ≥ 9.5
- B 8-9.4
- C 6.5-7.9
- D 5-6.4
- F <5

Código:

```
nota = float(input("Ingrese la nota: "))

if nota >= 9.5:
    print("Calificación: A")
elif 8 <= nota < 9.5:
    print("Calificación: B")
elif 6.5 <= nota < 8:
    print("Calificación: C")
elif 5 <= nota < 6.5:
    print("Calificación: D")
else:
    print("Calificación: F")
```

Explicación:

- Se usan rangos encadenados con `>=` y `<` para precisión.

Comandos:

- Variables: `nota`.
- Operadores: `>=`, `<=`, `<`.
- Funciones: `input()`, `float()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 33: Verificar adulto con nacionalidad

Enunciado:
Comprueba si una persona es mayor de edad (≥18) y tiene nacionalidad "española".

Código:

```
edad = int(input("Edad: "))
nacionalidad = input("Nacionalidad: ").lower()

if edad >= 18 and nacionalidad == "española":
    print("Adulto español")
else:
    print("No cumple la condición")
```

Explicación:

- `and` requiere que ambas condiciones sean verdaderas.

Comandos:

- Variables: `edad`, `nacionalidad`.
- Operadores: `>=`, `==`, `and`.
- Funciones: `input()`, `int()`, `lower()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 34: Evaluar temperatura

Enunciado:
Clasifica la temperatura: fría (<15), templada (15-25), caliente (>25).

Código:

```
temp = float(input("Ingrese la temperatura: "))


if temp < 15:
    print("Fría")
elif 15 <= temp <= 25:
    print("Templada")
else:
    print("Caliente")
```

Explicación:

- Comparaciones encadenadas permiten definir rango intermedio.

Comandos:

- Variables: `temp`.
- Operadores: `<`, `<=`, `>=`.
- Funciones: `input()`, `float()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 35: Número divisible por 2, 3 o ambos

Enunciado:
Verifica si un número es divisible por 2, 3, o ambos.

Código:

```
num = int(input("Ingrese un número: "))


if num % 2 == 0 and num % 3 == 0:
    print("Divisible por 2 y 3")
elif num % 2 == 0:
    print("Divisible solo por 2")
elif num % 3 == 0:
    print("Divisible solo por 3")
else:
    print("No divisible por 2 ni 3")
```

Explicación:

- Se usan condiciones anidadas con `and`.
- `elif` permite evaluar exclusividad de divisibilidad.

Comandos:

- Variables: `num`.
- Operadores: `%`, `==`, `and`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 36: Determinar triángulo válido

Enunciado:
Verifica si tres lados pueden formar un triángulo (suma de dos lados > tercer lado).

Código:

```
a = float(input("Lado a: "))
b = float(input("Lado b: "))
c = float(input("Lado c: "))

if a + b > c and a + c > b and b + c > a:
    print("Triángulo válido")
else:
    print("No es un triángulo")
```

Explicación:

- `and` combina las tres condiciones de desigualdad triangular.

Comandos:

- Variables: `a`, `b`, `c`.
- Operadores: `+`, `>`, `and`.
- Funciones: `input()`, `float()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 37: Determinar si año es bisiesto

Enunciado:
Verifica si un año es bisiesto: divisible por 4 y no por 100, salvo que sea divisible por 400.

Código:

```
año = int(input("Ingrese un año: "))


if (año % 4 == 0 and año % 100 != 0) or (año % 400 == 0):
    print("Es bisiesto")
else:
    print("No es bisiesto")
```

Explicación:

- Combinación de `and` y `or` para regla completa de bisiesto.

Comandos:

- Variables: `año`.
- Operadores: `%`, `==`, `!=`, `and`, `or`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 38: Aceptar o rechazar entrada

Enunciado:
Acepta solo números entre 1 y 100, múltiplo de 5.

Código:

```
num = int(input("Ingrese un número: "))


if 1 <= num <= 100 and num % 5 == 0:
    print("Número válido")
else:
    print("Número inválido")
```

Explicación:

- `and` combina rango y divisibilidad.

Comandos:

- Variables: `num`.
- Operadores: `<=`, `>=`, `%`, `==`, `and`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 39: Clasificación de altura

Enunciado:
Clasifica altura en baja (<1.60 m), media (1.60-1.80 m), alta (>1.80 m).

Código:

```
altura = float(input("Ingrese la altura en metros: "))


if altura < 1.60:
    print("Baja")
elif 1.60 <= altura <= 1.80:
    print("Media")
else:
    print("Alta")
```

Explicación:

- Rangos encadenados con `elif` permiten clasificar altura.

Comandos:

- Variables: `altura`.
- Operadores: `<`, `<=`, `>=`.
- Funciones: `input()`, `float()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 40: Edad para carnet de conducir especial

Enunciado:
Verifica si alguien puede obtener un carnet especial (18-70) y no tiene multas (`multas = 0`).

Código:

```
edad = int(input("Edad: "))
multas = int(input("Número de multas: "))

if 18 <= edad <= 70 and multas == 0:
    print("Puede obtener el carnet especial")
else:
    print("No cumple los requisitos")
```

Explicación:

- `and` combina rango de edad y condición de multas.

Comandos:

- Variables: `edad`, `multas`.
- Operadores: `<=`, `>=`, `==`, `and`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `else`.

Ejercicio 41: Determinar el trimestre académico

Enunciado:
Según el mes (1-12), indica el trimestre académico:

- Enero-marzo: 1er trimestre
- Abril-junio: 2º trimestre
- Julio-septiembre: 3er trimestre
- Octubre-diciembre: 4º trimestre

Código:

```
mes = int(input("Ingrese el mes (1-12): "))

if 1 <= mes <= 3:
    print("1er trimestre")
elif 4 <= mes <= 6:
    print("2º trimestre")
elif 7 <= mes <= 9:
    print("3er trimestre")
elif 10 <= mes <= 12:
    print("4º trimestre")
else:
    print("Mes inválido")
```

Explicación:

- `elif` permite evaluar múltiples rangos consecutivos.

Comandos:

- Variables: `mes`.
- Operadores: `<=`, `>=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 42: Determinar día de la semana

Enunciado:
Recibe un número (1-7) y devuelve el día de la semana.

Código:

```
num = int(input("Ingrese un número (1-7): "))


if num == 1:
    print("Lunes")
elif num == 2:
    print("Martes")
elif num == 3:
    print("Miércoles")
elif num == 4:
    print("Jueves")
elif num == 5:
    print("Viernes")
elif num == 6:
    print("Sábado")
elif num == 7:
    print("Domingo")
else:
    print("Número inválido")
```

Explicación:

- `elif` permite evaluar cada opción individualmente.

Comandos:

- Variables: `num`.
- Operadores: `==`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 43: Aceptar o rechazar acceso

Enunciado:
Valida usuario y contraseña. Ambos deben ser correctos.

Código:

```
usuario = input("Usuario: ")
contraseña = input("Contraseña: ")

if usuario == "admin":
    if contraseña == "1234":
        print("Acceso concedido")
    else:
        print("Contraseña incorrecta")
else:
    print("Usuario incorrecto")
```

Explicación:

- Condicional anidada permite diferenciar error de usuario y contraseña.

Comandos:

- Variables: `usuario`, `contraseña`.
- Operadores: `==`.
- Funciones: `input()`, `print()`.
- Estructuras: `if` anidado, `else`.

Ejercicio 44: Clasificación de edades múltiple

Enunciado:
Clasifica edades: niño (<12), adolescente (12-17), joven (18-30), adulto (31-60), mayor (>60).

Código:

```
edad = int(input("Ingrese la edad: "))


if edad < 12:
    print("Niño")
elif edad <= 17:
    print("Adolescente")
elif edad <= 30:
    print("Joven")
elif edad <= 60:
    print("Adulto")
else:
    print("Mayor")
```

Explicación:

- Condiciones encadenadas permiten varios rangos.

Comandos:

- Variables: `edad`.
- Operadores: `<`, `<=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 45: Clasificación de notas con recuperación

Enunciado:
Notas:

- ≥9: Excelente
- 7-8.9: Bueno
- 5-6.9: Regular
- <5: Insuficiente

Código:

```
nota = float(input("Ingrese la nota: "))


if nota >= 9:
    print("Excelente")
elif 7 <= nota < 9:
    print("Bueno")
elif 5 <= nota < 7:
    print("Regular")
else:
    print("Insuficiente")
```

Explicación:

- Condiciones intermedias con `<=` y `<`.

Comandos:

- Variables: `nota`.
- Operadores: `>=`, `<=`, `<`.
- Funciones: `input()`, `float()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 46: Número divisible por 2, 3 y 5

Enunciado:
Verifica divisibilidad múltiple y específica.

Código:

```
num = int(input("Ingrese un número: "))


if num % 2 == 0 and num % 3 == 0 and num % 5 == 0:
    print("Divisible por 2, 3 y 5")
elif num % 2 == 0 and num % 3 == 0:
    print("Divisible por 2 y 3")
elif num % 2 == 0:
    print("Divisible solo por 2")
else:
    print("No divisible por 2 ni 3 ni 5")
```

Explicación:

- `and` combina varias condiciones para comprobar múltiplos simultáneos.

Comandos:

- Variables: `num`.
- Operadores: `%`, `==`, `and`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 47: Validación de contraseña fuerte

Enunciado:
Contraseña válida si tiene ≥8 caracteres y contiene al menos un número.

Código:

```
contraseña = input("Ingrese la contraseña: ")


if len(contraseña) >= 8:
    if any(c.isdigit() for c in contraseña):
        print("Contraseña válida")
    else:
        print("Debe contener al menos un número")
else:
    print("Debe tener al menos 8 caracteres")
```

Explicación:

- `len()` obtiene longitud.
- `any()` verifica si algún carácter es dígito.
- Condicional anidada para validar varias reglas.

Comandos:

- Variables: `contraseña`, `c`.
- Operadores: `>=`.
- Funciones: `input()`, `len()`, `any()`, `isdigit()`, `print()`.
- Estructuras: `if` anidado, `else`.

Ejercicio 48: Mayor de tres números con igualdad

Enunciado:
Determina el mayor de tres números y si hay empate.

Código:

```
a = int(input("Número 1: "))
b = int(input("Número 2: "))
c = int(input("Número 3: "))

if a == b == c:
    print("Todos son iguales")
elif a >= b and a >= c:
    print("El mayor es", a)
elif b >= a and b >= c:
    print("El mayor es", b)
else:
    print("El mayor es", c)
```

Explicación:

- `==` encadenado detecta igualdad total.
- `>=` permite encontrar el mayor.

Comandos:

- Variables: `a`, `b`, `c`.
- Operadores: `==`, `>=`, `and`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 49: Clasificación de temperatura

Enunciado:
Temperatura en Celsius:

- <0: Congelante
- 0-15: Fría
- 16-25: Templada
- 26-35: Caliente
    - 35: Muy caliente

Código:

```
temp = float(input("Ingrese la temperatura: "))


if temp < 0:
    print("Congelante")
elif 0 <= temp <= 15:
    print("Fría")
elif 16 <= temp <= 25:
    print("Templada")
elif 26 <= temp <= 35:
    print("Caliente")
else:
    print("Muy caliente")
```

Explicación:

- Rangos encadenados permiten clasificar temperatura en cinco categorías.

Comandos:

- Variables: `temp`.
- Operadores: <, <=, >=.
- Funciones: `input()`, `float()`, `print()`.
- Estructuras: `if`, `elif`, `else`.

Ejercicio 50: Acceso según edad y membresía

Enunciado:
Acceso permitido si edad ≥18 y tiene membresía activa.

Código:

```
edad = int(input("Edad: "))
membresia = input("Membresía activa (sí/no): ").lower()

if edad >= 18 and membresia == "sí":
    print("Acceso permitido")
else:
    print("Acceso denegado")
```

Explicación:

- `and` asegura que ambas condiciones sean verdaderas para permitir acceso.
- `.lower()` normaliza la entrada para comparación.

Comandos:

- Variables: `edad`, `membresia`.
- Operadores: `>=`, `==`, `and`.
- Funciones: `input()`, `int()`, `lower()`, `print()`.
- Estructuras: `if`, `else`.

MANUAL EXPLICATIVO DE EJERCICIOS DE ESTRUCTURAS REPETITIVAS

BLOQUE I: EJERCICIOS BÁSICOS CON `FOR` Y ACUMULACIÓN

Ejercicio 1: Contar del 1 al 10

- **Enunciado:** Escribe un programa que muestre los números del 1 al 10.
- **Explicación:** El comando `for` se utiliza para recorrer una secuencia. La función `range(1, 11)` genera los números enteros iniciando en 1 y terminando antes del 11, abarcando así del 1 al 10. La función `print(i)` se encarga de mostrar cada número.
- **Controles y Comandos:**
    - Variables de control: `i`.
    - Funciones: `print()`, `range()`.
    - Estructura: `for`.

Ejercicio 2: Contar del 10 al 1

- **Enunciado:** Muestra los números del 10 al 1 en orden descendente.
- **Explicación:** Se utiliza `range(10, 0, -1)`. Esta función inicia la cuenta en 10, establece el límite antes de llegar a 0, y define un paso de `-1` para asegurar el orden descendente.
- **Controles y Comandos:**
    - Variable de control: `i`.
    - Funciones: `print()`, `range()`.
    - Estructura: `for`.

Ejercicio 3: Suma de los primeros 10 números

- **Enunciado:** Calcula la suma de los números del 1 al 10.
- **Explicación:** Se requiere inicializar la variable `suma = 0`. Dentro del bucle, el operador `suma += i` se utiliza para acumular progresivamente los valores de la secuencia.
- **Controles y Comandos:**
    - Variables: `suma`, `i`.
    - Operadores: `+=`.
    - Funciones: `print()`, `range()`.
    - Estructura: `for`.

Ejercicio 4: Números pares del 1 al 20

- **Enunciado:** Muestra todos los números pares del 1 al 20.
- **Explicación:** Se utiliza una estructura `if` dentro del bucle para filtrar los números. La condición `i % 2 == 0` verifica si el número `i` es par, basándose en si el residuo de la división por 2 es cero.
- **Controles y Comandos:**
    - Variables: `i`.
    - Operadores: `%` (módulo), `==` (igualdad).

    - Funciones: `print()`, `range()`.
    - Estructuras: `for`, `if`.

Ejercicio 5: Suma de números pares hasta N

- **Enunciado:** Pide un número N y suma todos los pares hasta N.
- **Explicación:** El rango se define como `range(2, N+1, 2)`, lo cual garantiza que el bucle recorra **solo los números pares**. El operador `suma += i` se encarga de la acumulación de estos pares.
- **Controles y Comandos:**
    - Variables: `N`, `suma`, `i`.
    - Operadores: `+=`.
    - Funciones: `input()` (solicita datos), `int()` (convierte a entero), `print()`, `range()`.
    - Estructura: `for`.

Ejercicio 6: Factorial de un número

- **Enunciado:** Calcula el factorial de un número n usando un bucle.
- **Explicación:** El factorial se calcula multiplicando números sucesivos, lo cual se logra con el operador de multiplicación y asignación: `factorial *= i`.
- **Controles y Comandos:**
    - Variables: `n`, `factorial`, `i`.
    - Operadores: `*=`.
    - Funciones: `input()`, `int()`, `print()`, `range()`.
    - Estructura: `for`.

Ejercicio 7: Tabla de multiplicar

- **Enunciado:** Muestra la tabla de multiplicar de un número ingresado.
- **Explicación:** El bucle recorre los números del 1 al 10, multiplicando el número ingresado (`n`) por cada iterador (`i`).
- **Controles y Comandos:**
    - Variables: `n`, `i`.
    - Operadores: `*` (multiplicación).
    - Funciones: `input()`, `int()`, `print()`, `range()`.
    - Estructura: `for`.

BLOQUE II: EJERCICIOS CON `WHILE` Y MANEJO DE ENTRADA

Ejercicio 8: Números impares usando while

- **Enunciado:** Muestra los números impares del 1 al 15 usando `while`.
- **Explicación:** La estructura `while` mantiene el bucle activo siempre y cuando la condición `i <= 15` sea verdadera. Es fundamental el comando `i += 1` para incrementar el contador y evitar un bucle infinito. Dentro del bucle, `if i % 2 != 0` filtra los impares.

- **Controles y Comandos:**
    - Variables: `i`.
    - Operadores: `%`, `!=` (diferente de), `<=` (menor o igual), `+=`.
    - Funciones: `print()`.
    - Estructuras: `while`, `if`.

Ejercicio 9: Suma hasta que el usuario ingrese 0

- **Enunciado:** Suma números ingresados por el usuario hasta que ingrese 0.
- **Explicación:** El bucle `while num != 0` se ejecuta repetidamente hasta que el usuario introduce el valor de terminación (0). Es crucial que la variable `num` se solicite y actualice dentro del bucle para que la condición de salida pueda evaluarse.
- **Controles y Comandos:**
    - Variables: `suma`, `num`.
    - Operadores: `!=`, `+=`.
    - Funciones: `input()`, `int()`, `print()`.
    - Estructura: `while`.

Ejercicio 10: Contar dígitos de un número

- **Enunciado:** Cuenta cuántos dígitos tiene un número entero positivo.
- **Explicación:** El bucle se mantiene mientras el número (`num`) sea mayor que 0. Dentro del bucle, el operador de división entera con asignación (`num //= 10`) elimina el último dígito en cada paso. Simultáneamente, la variable `contador` acumula la cantidad de veces que se realizó esta operación, resultando en el número de dígitos.
- **Controles y Comandos:**
    - Variables: `num`, `contador`.
    - Operadores: `//=`, `+=`, `>` (mayor que).
    - Funciones: `input()`, `int()`, `print()`.
    - Estructura: `while`.

BLOQUE III: EJERCICIOS AVANZADOS DE FILTRADO Y CADENAS

Ejercicio 11: Suma de números pares e impares

- **Enunciado:** Calcula la suma de números pares e impares del 1 al 20 por separado.
- **Explicación:** Se utiliza una estructura `if/else` dentro del bucle. Si la condición `i % 2 == 0` (par) se cumple, se acumula en `suma_pares`. Si no se cumple, el bloque `else` captura los números impares y los acumula en `suma_impares`.
- **Controles y Comandos:**
    - Variables: `i`, `suma_pares`, `suma_impares`.
    - Operadores: `%`, `==`, `+=`.
    - Funciones: `print()`, `range()`.
    - Estructuras: `for`, `if`, `else`.

Ejercicio 12: Números primos hasta N

- **Enunciado:** Muestra todos los números primos hasta un número N.
- **Explicación:** Se utiliza un **bucle anidado**. El bucle interno (`for i in range(2, int(num**0.5)+1)`) busca divisores, pero solo hasta la raíz cuadrada del número para optimizar. Si se encuentra un divisor (`num % i == 0`), se establece la bandera `primo = False` y la instrucción `break` detiene la búsqueda de divisores para ese número específico.
- **Controles y Comandos:**
    - Variables: `N`, `num`, `i`, `primo`.
    - Operadores: `%`, `==`, `**` (potencia), `+`.
    - Funciones: `input()`, `int()`, `print()`.
    - Estructuras: `for`, `if`, `break`.

Ejercicio 13: Contar vocales en una cadena

- **Enunciado:** Cuenta cuántas vocales tiene una palabra ingresada por el usuario.
- **Explicación:** Se usa el método `.lower()` inmediatamente después de la entrada para convertir toda la palabra a minúsculas y asegurar que la comparación sea efectiva para todas las vocales. El operador `in "aeiou"` verifica si el carácter actual (`letra`) está contenido en la secuencia de vocales.
- **Controles y Comandos:**
    - Variables: `palabra`, `letra`, `contador`.
    - Operadores: `in`, `+=`.
    - Funciones: `input()`, `lower()`, `print()`.
    - Estructuras: `for`, `if`.

Ejercicio 14: Números múltiplos de 3

- **Enunciado:** Muestra los números múltiplos de 3 entre 1 y 50.
- **Explicación:** Se recorre el rango del 1 al 50. La condición `% 3 == 0` actúa como filtro, mostrando solo aquellos números cuyo módulo al dividirse por 3 es cero.
- **Controles y Comandos:**
    - Variables: `i`.
    - Operadores: `%`, `==`.
    - Funciones: `print()`, `range()`.
    - Estructuras: `for`, `if`.

Ejercicio 15: Contar caracteres de una cadena

- **Enunciado:** Cuenta cuántos caracteres tiene una cadena ingresada por el usuario.
- **Explicación:** El bucle `for c in texto` itera sobre cada carácter (`c`) presente en la variable `texto`. El contador se incrementa (`contador += 1`) en cada iteración, obteniendo el total de caracteres.
- **Controles y Comandos:**
    - Variables: `texto`, `c`, `contador`.
    - Operadores: `+=`.

- Funciones: `input()`, `print()`.
- Estructura: `for`.

Ejercicio 16: Tabla de multiplicar múltiple

- **Enunciado:** Muestra las tablas de multiplicar del 1 al 5.
- **Explicación:** Este ejercicio utiliza un **bucle anidado**. El bucle externo (`for i`) selecciona la tabla (del 1 al 5), y el bucle interno (`for j`) realiza las multiplicaciones del 1 al 10 para esa tabla. La función `print()` vacía se utiliza para separar visualmente las tablas.
- **Controles y Comandos:**
  - Variables: `i`, `j`.
  - Operadores: `*`.
  - Funciones: `print()`, `range()`.
  - Estructuras: `for anidado`.

Ejercicio 17: Sumar dígitos de un número

- **Enunciado:** Suma los dígitos de un número ingresado.
- **Explicación:** Dentro del bucle `while num > 0`, se usa `% 10` para obtener el último dígito del número, el cual se suma al acumulador. Inmediatamente, se usa `// 10` (división entera) para eliminar ese último dígito y preparar el número para la siguiente iteración.
- **Controles y Comandos:**
  - Variables: `num`, `suma`.
  - Operadores: `%`, `//` (división entera), `+=`, `>`.
  - Funciones: `input()`, `int()`, `print()`.
  - Estructura: `while`.

Ejercicio 18: Invertir una cadena

- **Enunciado:** Muestra la cadena ingresada al revés.
- **Explicación:** Se recorre cada carácter (`c`) de la cadena original. La clave es la concatenación `invertido = c + invertido`, que antepone el carácter actual al inicio de la cadena invertida que se está construyendo.
- **Controles y Comandos:**
  - Variables: `texto`, `c`, `invertido`.
  - Operadores: `+` (concatenación).
  - Funciones: `input()`, `print()`.
  - Estructura: `for`.

Ejercicio 19: Contar letras y números

- **Enunciado:** Cuenta cuántas letras y números hay en una cadena.
- **Explicación:** Se itera sobre cada carácter (`c`). La función `.isalpha()` se utiliza para detectar si el carácter es una letra, e `.isdigit()` se utiliza para detectar si es un dígito. Se usa `if/elif` para clasificar y contar.
- **Controles y Comandos:**
  - Variables: `texto`, `c`, `letras`, `numeros`.
  - Operadores: `+=`.

- Funciones: `input()`, `isalpha()`, `isdigit()`, `print()`.
- Estructuras: `for`, `if`, `elif`.

Ejercicio 20: Patrón de asteriscos

- **Enunciado:** Muestra un triángulo de asteriscos con 5 filas.
- **Explicación:** El bucle itera de 1 a 5. El operador de repetición de cadenas `*` multiplica el string `"*"` por el valor del contador `i` en cada iteración, generando el patrón creciente.
- **Controles y Comandos:**
  - Variables: `i`.
  - Operadores: `*` (repetición de cadenas).
  - Funciones: `print()`, `range()`.
  - Estructura: `for`.

BLOQUE IV: EJERCICIOS CON LISTAS Y ANÁLISIS DE DATOS

Ejercicio 21: Suma de números ingresados en lista

- **Enunciado:** Pide al usuario 5 números, guárdalos en una lista y muestra la suma total.
- **Explicación:** Se utiliza un primer bucle para solicitar los 5 números y la función `append()` para añadirlos a la lista `numeros`. Un segundo bucle recorre la lista ya creada para calcular la `suma` total.
- **Controles y Comandos:**
  - Variables: `numeros`, `i`, `n`, `num`, `suma`.
  - Operadores: `+=`.
  - Funciones: `input()`, `int()`, `print()`, `append()`, `range()`.
  - Estructura: `for`.

Ejercicio 22: Encontrar el mayor de una lista

- **Enunciado:** Pide 5 números, guárdalos en una lista y muestra el mayor.
- **Explicación:** Después de llenar la lista, la variable `mayor` se inicializa con el primer elemento (`numeros`). El segundo bucle compara cada número de la lista (`num`) con el valor actual de `mayor`; si `num > mayor`, se actualiza el valor de `mayor`.
- **Controles y Comandos:**
  - Variables: `numeros`, `i`, `n`, `num`, `mayor`.
  - Operadores: `>` (mayor que).
  - Funciones: `input()`, `int()`, `print()`, `append()`, `range()`.
  - Estructuras: `for`, `if`.

Ejercicio 23: Promedio de lista

- **Enunciado:** Pide 5 números, calcula y muestra el promedio.
- **Explicación:** Se suma el total de los elementos de la lista. La función `len(numeros)` devuelve la cantidad de elementos, y el promedio se calcula dividiendo la suma entre esa cantidad.
- **Controles y Comandos:**

    - Variables: `numeros`, `i`, `n`, `num`, `suma`, `promedio`.
    - Operadores: `+=`, `/` (división).
    - Funciones: `input()`, `float()` (para permitir decimales), `print()`, `append()`, `len()`, `range()`.
    - Estructura: `for`.

Ejercicio 24: Contar números positivos y negativos

- **Enunciado:** Pide 10 números y cuenta cuántos son positivos y cuántos negativos.
- **Explicación:** Se utiliza la estructura `if/elif` dentro del bucle de entrada para clasificar cada número `n` al momento de ser ingresado. `if n > 0` cuenta los positivos, y `elif n < 0` cuenta los negativos.
- **Controles y Comandos:**
    - Variables: `positivos`, `negativos`, `i`, `n`.
    - Operadores: `>`, `<`, `+=`.
    - Funciones: `input()`, `int()`, `print()`, `range()`.
    - Estructuras: `for`, `if`, `elif`.

Ejercicio 25: Suma de números pares de lista

- **Enunciado:** Pide 10 números, guárdalos en una lista y suma solo los pares.
- **Explicación:** Una vez que la lista está completa, se recorre con un bucle. La condición `if num % 2 == 0` identifica cuáles elementos de la lista son pares, y solo esos se suman a `suma_pares`.
- **Controles y Comandos:**
    - Variables: `numeros`, `i`, `n`, `num`, `suma_pares`.
    - Operadores: `%`, `==`, `+=`.
    - Funciones: `input()`, `int()`, `print()`, `append()`, `range()`.
    - Estructuras: `for`, `if`.

Ejercicio 26: Invertir lista

- **Enunciado:** Pide 5 números, guárdalos en una lista y muestra la lista invertida.
- **Explicación:** Para recorrer la lista en orden inverso, se usa `range(len(numeros)-1, -1, -1)`. Esto establece el inicio en el último índice, el final antes del -1 (es decir, incluye el índice 0), y un paso de -1. Los elementos se añaden a una nueva lista, `invertida`, siguiendo este orden.
- **Controles y Comandos:**
    - Variables: `numeros`, `invertida`, `i`, `n`.
    - Operadores: `-`, `append()`.
    - Funciones: `input()`, `int()`, `print()`, `len()`, `range()`.
    - Estructura: `for`.

Ejercicio 27: Contar vocales en lista de palabras

- **Enunciado:** Pide 3 palabras, cuenta las vocales de cada palabra.

- **Explicación:** Este ejercicio utiliza un **bucle anidado**. El bucle externo itera sobre cada `palabra` en la lista, y el bucle interno itera sobre cada `letra` de esa palabra. `in "aeiou"` verifica si el carácter es una vocal.
- **Controles y Comandos:**
    - Variables: `palabras`, `i`, `palabra`, `letra`, `contador`.
    - Operadores: `in`, `+=`.
    - Funciones: `input()`, `lower()`, `print()`, `append()`, `range()`.
    - Estructuras: `for`, `if`.

Ejercicio 28: Números pares e impares en lista

- **Enunciado:** Pide 10 números y muestra cuántos son pares y cuántos impares.
- **Explicación:** Después de almacenar todos los números en la lista, se utiliza un bucle para recorrerla. Se usa `if/else` para clasificar si cada número tiene un módulo 0 (`pares`) o si no lo tiene (`impares`), incrementando los contadores respectivos.
- **Controles y Comandos:**
    - Variables: `numeros`, `i`, `n`, `num`, `pares`, `impares`.
    - Operadores: `%`, `==`, `+=`.
    - Funciones: `input()`, `int()`, `print()`, `append()`, `range()`.
    - Estructuras: `for`, `if`, `else`.

Ejercicio 29: Suma de elementos positivos de lista

- **Enunciado:** Pide 8 números, guarda en lista y suma solo los positivos.
- **Explicación:** Se recorre la lista completa de números. La condición `if num > 0` filtra los elementos, asegurando que solo los números mayores que cero se acumulen en `suma_positivos`.
- **Controles y Comandos:**
    - Variables: `numeros`, `i`, `n`, `num`, `suma_positivos`.
    - Operadores: `>`, `+=`.
    - Funciones: `input()`, `int()`, `print()`, `append()`, `range()`.
    - Estructuras: `for`, `if`.

BLOQUE V: PATRONES Y VALIDACIÓN CON `WHILE`

Ejercicio 30: Patrón de asteriscos cuadrado

- **Enunciado:** Muestra un cuadrado de asteriscos de 5x5 usando bucles anidados.
- **Explicación:** Se utiliza un **bucle anidado**. El bucle externo (`i`) controla las filas, y el bucle interno (`j`) controla la impresión de columnas. El parámetro `end=" "` en `print("*", end=" ")` es vital, ya que evita el salto de línea para cada columna, permitiendo que los asteriscos formen una fila. El `print()` vacío después del bucle interno genera el salto de línea para pasar a la siguiente fila.
- **Controles y Comandos:**
    - Variables: `i`, `j`.

    - Funciones: `print()`, `range()`.
    - Estructuras: `for anidado`.

Ejercicio 31: Validar entrada positiva

- **Enunciado:** Pide al usuario un número positivo; si ingresa un número negativo, sigue pidiendo hasta que sea positivo.
- **Explicación:** Este es un ejemplo de bucle de validación. La condición `while num < 0` fuerza la repetición de la solicitud de entrada. El bucle solo termina cuando se ingresa un número no negativo.
- **Controles y Comandos:**
    - Variables: `num`.
    - Operadores: `<`.
    - Funciones: `input()`, `int()`, `print()`.
    - Estructura: `while`.

Ejercicio 32: Sumar números hasta que se ingrese 0

- **Enunciado:** Suma todos los números ingresados por el usuario hasta que ingrese 0.
- **Explicación:** Se usa `while True` para crear un bucle que se ejecuta indefinidamente. La terminación se gestiona internamente: si el número ingresado es 0 (`if num == 0`), la instrucción `break` detiene el bucle.
- **Controles y Comandos:**
    - Variables: `suma`, `num`.
    - Operadores: `==`, `+=`.
    - Funciones: `input()`, `int()`, `print()`.
    - Estructuras: `while`, `if`, `break`.

Ejercicio 33: Factorial con while

- **Enunciado:** Calcula el factorial de un número usando `while`.
- **Explicación:** Es una implementación del factorial usando `while`. Se inicializa un contador `i = 1`. El bucle continúa mientras `i <= n`. La operación `factorial *= i` acumula el producto, y `i += 1` asegura que se procesen todos los números hasta `n`.
- **Controles y Comandos:**
    - Variables: `n`, `factorial`, `i`.
    - Operadores: `<=`, `*=`, `+=`.
    - Funciones: `input()`, `int()`, `print()`.
    - Estructura: `while`.

Ejercicio 34: Contar dígitos pares e impares

- **Enunciado:** Cuenta cuántos dígitos pares e impares tiene un número entero positivo.
- **Explicación:** Mientras el número sea mayor que 0, se usa `digito = num % 10` para obtener el último dígito. Luego, se clasifica si el `digito` es par (`digito % 2 == 0`) o impar. Finalmente, `num //= 10` elimina el dígito analizado.
- **Controles y Comandos:**
    - Variables: `num`, `digito`, `pares`, `impares`.

- Operadores: `%`, `//`, `==`, `+=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `while`, `if`, `else`.

Ejercicio 35: Patrón pirámide

- **Enunciado:** Muestra una pirámide de asteriscos de 5 filas.
- **Explicación:** La creación de la pirámide se basa en una fórmula de impresión. `" " * (5-i)` genera la cantidad decreciente de espacios iniciales para centrar el patrón. Por su parte, `"* " * (2*i-1)` genera la cantidad creciente y par de asteriscos necesaria para formar la estructura triangular.
- **Controles y Comandos:**
  - Variables: `i`.
  - Operadores: `*` (repetición), `-`.
  - Funciones: `print()`, `range()`.
  - Estructura: `for`.

Ejercicio 36: Número primo usando while

- **Enunciado:** Verifica si un número ingresado es primo usando `while`.
- **Explicación:** Se realiza una verificación preliminar para números menores a 2. El bucle `while i <= num // 2` revisa los posibles divisores únicamente hasta la mitad del número. Si `num % i == 0`, se concluye que no es primo, se establece `primo = False` y `break` detiene el bucle de búsqueda inmediatamente.
- **Controles y Comandos:**
  - Variables: `num`, `i`, `primo`.
  - Operadores: `%`, `<=`, `==`, `+=`.
  - Funciones: `input()`, `int()`, `print()`.
  - Estructuras: `if`, `while`, `break`.

Ejercicio 37: Contar ocurrencias de una letra

- **Enunciado:** Cuenta cuántas veces aparece una letra en una palabra.
- **Explicación:** Las entradas se normalizan a minúsculas usando `.lower()`. El bucle `for letra in palabra` itera sobre cada carácter. La condición `if letra == letra_buscar` compara el carácter actual con la letra deseada, incrementando el contador si son iguales.
- **Controles y Comandos:**
  - Variables: `palabra`, `letra_buscar`, `letra`, `contador`.
  - Operadores: `==`, `+=`.
  - Funciones: `input()`, `lower()`, `print()`.
  - Estructuras: `for`, `if`.

Ejercicio 38: Patrón cuadrado con diagonales

- **Enunciado:** Muestra un cuadrado 5x5 de asteriscos con diagonales vacías.

- **Explicación:** Este patrón requiere un **bucle anidado** para gestionar coordenadas (`i` para filas, `j` para columnas). El asterisco se imprime si se cumplen varias condiciones combinadas con el operador lógico `or`:
    1. Estar en los bordes (`i == 0 or i == 4` o `j == 0 or j == 4`).
    2. Estar en la diagonal principal (`i == j`).
    3. Estar en la diagonal secundaria (`i + j == 4`). Si ninguna de estas condiciones se cumple, se imprime un espacio.
- **Controles y Comandos:**
    1. Variables: `i`, `j`.
    2. Operadores: `==`, `+`.
    3. Funciones: `print()`, `range()`.
    4. Estructuras: `for anidado`, `if`, `else`.

Ejercicio 39: Promedio con validación

- **Enunciado:** Pide al usuario números hasta que ingrese 0, y muestra el promedio de los ingresados (excepto 0).
- **Explicación:** Se usa `while True` y `break` cuando se ingresa 0. Se utiliza la variable `contador` para llevar la cuenta de los números válidos. La validación final (`if contador > 0`) es esencial para evitar un error de división por cero si el usuario solo ingresó 0 o no ingresó números.
- **Controles y Comandos:**
    - Variables: `suma`, `contador`, `n`, `promedio`.
    - Operadores: `+=`, `/`, `==`, `>`.
    - Funciones: `input()`, `int()`, `print()`.
    - Estructuras: `while`, `if`, `break`, `else`.

Ejercicio 40: Patrón de triángulo invertido

- **Enunciado:** Muestra un triángulo invertido de asteriscos de 5 filas.
- **Explicación:** Se utiliza `range(5, 0, -1)` para recorrer en orden descendente (de 5 a 1). El operador de repetición `*` imprime `i` asteriscos, creando un patrón que disminuye en cada fila.
- **Controles y Comandos:**
    - Variables: `i`.
    - Operadores: `*`, `-`.
    - Funciones: `print()`, `range()`.
    - Estructura: `for`.

Ejercicio 41: Patrón de pirámide invertida

- **Enunciado:** Muestra una pirámide invertida de asteriscos de 5 filas.
- **Explicación:** Similar al ejercicio 35, pero con un rango descendente (`range(5, 0, -1)`). La fórmula `" " * (5 - i)` calcula los espacios iniciales necesarios, que aumentan en cada fila. `"*" * (2*i - 1)` genera el número de asteriscos, que disminuye en cada fila.
- **Controles y Comandos:**

- Variables: `i`.
- Operadores: `*`, `-`.
- Funciones: `print()`, `range()`.
- Estructura: `for`.

Ejercicio 42: Suma de múltiplos de 3 y 5

- **Enunciado:** Calcula la suma de los números del 1 al 100 que son múltiplos de 3 o 5.
- **Explicación:** Se recorren los números del 1 al 100. La condición `if i % 3 == 0 or i % 5 == 0` utiliza el operador lógico `or` para incluir un número en la suma si cumple **al menos una** de las dos condiciones de divisibilidad.
- **Controles y Comandos:**
  - Variables: `suma`, `i`.
  - Operadores: `%`, `==`, `or`, `+=`.
  - Funciones: `print()`, `range()`.
  - Estructuras: `for`, `if`.

Ejercicio 43: Validación de contraseña

- **Enunciado:** Pide una contraseña que cumpla con longitud >= 6 y contenga un número; sigue pidiendo hasta que sea válida.
- **Explicación:** Se usa `while True` para forzar la repetición hasta que la validación sea exitosa. La validación combina dos condiciones usando el operador `and`:
  1. `len(contraseña) >= 6` verifica que la longitud sea mayor o igual a 6.
  2. `any(c.isdigit() for c in contraseña)` verifica que al menos un carácter sea un dígito. Si ambas son verdaderas, se ejecuta `break`.
- **Controles y Comandos:**
  1. Variables: `contraseña`, `c`.
  2. Operadores: `>=`, `and`.
  3. Funciones: `input()`, `len()`, `print()`, `any()`, `isdigit()`.
  4. Estructuras: `while`, `if`, `break`.

Ejercicio 44: Patrón de números

- **Enunciado:** Muestra el siguiente patrón: 1, 12, 123, 1234, 12345.
- **Explicación:** Se utiliza un **bucle anidado**. El bucle externo (`i`) define cuántos números se imprimirán en total en la fila actual, y el bucle interno (`j`) imprime los números desde 1 hasta `i`. `end=""` evita que cada número imprima un salto de línea.
- **Controles y Comandos:**
  - Variables: `i`, `j`.
  - Operadores: `+`.
  - Funciones: `print()`, `range()`.
  - Estructuras: `for anidado`.

Ejercicio 45: Promedio de números positivos

- **Enunciado:** Pide números hasta que se ingrese 0 y muestra el promedio de los positivos.

- **Explicación:** Se utiliza `while True` con un `break` si `n == 0`. Solo si el número es positivo (`if n > 0`), se incrementan tanto la `suma` como el `contador`. La validación final (`if contador > 0`) previene el error de división por cero.
- **Controles y Comandos:**
    - Variables: `suma`, `contador`, `n`.
    - Operadores: `>`, `==`, `+=`, `/`.
    - Funciones: `input()`, `int()`, `print()`.
    - Estructuras: `while`, `if`, `break`.

Ejercicio 46: Tabla de multiplicar completa

- **Enunciado:** Muestra las tablas de multiplicar del 1 al 10.
- **Explicación:** Es un **bucle anidado** donde el bucle externo (`i`) recorre las tablas (1 a 10), y el interno (`j`) recorre los multiplicadores (1 a 10). Se utiliza un f-string para formatear la salida (`f"{i} x {j} = {i*j}"`).
- **Controles y Comandos:**
    - Variables: `i`, `j`.
    - Operadores: `*`.
    - Funciones: `print()`, `range()`.
    - Estructuras: `for anidado`.

Ejercicio 47: Patrón de diamante

- **Enunciado:** Muestra un diamante de asteriscos de 5 filas por lado.
- **Explicación:** El patrón se genera mediante dos bloques de bucles `for`. El primer bucle construye la **pirámide superior** (creciente). El segundo bucle (`range(n-1, 0, -1)`) construye la **pirámide inferior invertida** (decreciente), para completar la forma de diamante.
- **Controles y Comandos:**
    - Variables: `i`, `n`.
    - Operadores: `*`, `-`.
    - Funciones: `print()`, `range()`.
    - Estructuras: `for`.

Ejercicio 48: Contar letras y dígitos en texto

- **Enunciado:** Cuenta cuántas letras y dígitos hay en un texto ingresado.
- **Explicación:** Se recorre cada carácter `c` del texto. Se emplean los métodos `.isalpha()` para identificar letras y `.isdigit()` para identificar dígitos (números). Se usan contadores separados para `letras` y `digitos`.
- **Controles y Comandos:**
    - Variables: `texto`, `c`, `letras`, `digitos`.
    - Operadores: `+=`.
    - Funciones: `input()`, `isalpha()`, `isdigit()`, `print()`.
    - Estructuras: `for`, `if`, `elif`.

Ejercicio 49: Suma de diagonales en matriz

- **Enunciado:** Dada una matriz 3x3 de números ingresados por el usuario, suma las diagonales principal y secundaria.
- **Explicación:** Se utiliza un **bucle anidado** para ingresar los 9 elementos y construir la matriz. La suma se realiza en un segundo bucle:
    - La diagonal principal se suma usando `matriz[i][i]`.
    - La diagonal secundaria se suma usando `matriz[i][2-i]` (donde 2 es el índice máximo para una matriz 3x3).
- **Controles y Comandos:**
    - Variables: `matriz`, `fila`, `i`, `j`, `n`, `suma_principal`, `suma_secundaria`.
    - Operadores: `+=`.
    - Funciones: `input()`, `int()`, `print()`, `append()`, `range()`.
    - Estructuras: `for anidado`.

Ejercicio 50: Patrón de números en espiral

- **Enunciado:** Muestra un patrón 4x4 de números del 1 al 16 en forma de espiral (simulado con bucles simples).
- **Explicación:** La simulación se logra creando primero una lista con los números del 1 al 16. Luego, se usa un **bucle anidado** (4x4) para acceder a estos números. La posición en la lista se calcula mediante la fórmula `numeros[i*4+j]`, que selecciona los elementos de manera secuencial por fila. El formato `:2` en el f-string (`{...:2}`) se emplea para alinear los números en la salida.
- **Controles y Comandos:**
    - Variables: `numeros`, `i`, `j`.
    - Operadores: `*`, `+`.
    - Funciones: `list()`, `range()`, `print()`.
    - Estructuras: `for anidado`.

Ejercicio 1: Contar del 1 al 10

Enunciado:
Escribe un programa que muestre los números del 1 al 10.

Código:

```
for i in range(1, 11):
    print(i)
```

Explicación:

- `for` recorre una secuencia.
- `range(1, 11)` genera los números del 1 al 10 (11 no incluido).
- `print(i)` muestra cada número.

Comandos:

- Variable de control: `i`.
- Funciones: `print()`, `range()`.
- Estructura: `for`.

Ejercicio 2: Contar del 10 al 1

Enunciado:
Muestra los números del 10 al 1 en orden descendente.

Código:

```
for i in range(10, 0, -1):
    print(i)
```

Explicación:

- `range(10, 0, -1)` empieza en 10, termina antes de 0, con paso -1.

Comandos:

- Variable de control: `i`.
- Funciones: `print()`, `range()`.
- Estructura: `for`.

Ejercicio 3: Suma de los primeros 10 números

Enunciado:
Calcula la suma de los números del 1 al 10.

Código:

```
suma = 0
for i in range(1, 11):
    suma += i
print("La suma es:", suma)
```

Explicación:

- `suma += i` acumula los valores.
- Se inicializa `suma = 0`.

Comandos:

- Variables: `suma`, `i`.
- Operadores: `+=`.
- Funciones: `print()`, `range()`.
- Estructura: `for`.

Ejercicio 4: Números pares del 1 al 20

Enunciado:
Muestra todos los números pares del 1 al 20.

Código:

```
for i in range(1, 21):
    if i % 2 == 0:
        print(i)
```

Explicación:

- `i % 2 == 0` verifica si `i` es par.
- `if` dentro del bucle filtra los números.

Comandos:

- Variables: `i`.
- Operadores: `%`, `==`.
- Funciones: `print()`, `range()`.
- Estructuras: `for`, `if`.

Ejercicio 5: Suma de números pares hasta N

Enunciado:
Pide un número `N` y suma todos los pares hasta `N`.

Código:

```
N = int(input("Ingrese un número: "))
suma = 0
for i in range(2, N+1, 2):
    suma += i
print("Suma de pares hasta", N, ":", suma)
```

Explicación:

- `range(2, N+1, 2)` recorre solo los pares.
- `suma += i` acumula los pares.

Comandos:

- Variables: `N`, `suma`, `i`.
- Operadores: `+=`.
- Funciones: `input()`, `int()`, `print()`, `range()`.
- Estructuras: `for`.

Ejercicio 6: Factorial de un número

Enunciado:
Calcula el factorial de un número `n` usando un bucle.

Código:

```
n = int(input("Ingrese un número: "))
factorial = 1
for i in range(1, n+1):
    factorial *= i
print("Factorial de", n, "es", factorial)
```

Explicación:

- `factorial *= i` multiplica los números sucesivos.

Comandos:

- Variables: `n`, `factorial`, `i`.
- Operadores: `*=`.
- Funciones: `input()`, `int()`, `print()`, `range()`.
- Estructuras: `for`.

Ejercicio 7: Tabla de multiplicar

Enunciado:
Muestra la tabla de multiplicar de un número ingresado.

Código:

```
n = int(input("Ingrese un número: "))
for i in range(1, 11):
    print(n, "x", i, "=", n*i)
```

Explicación:

- Multiplica `n` por cada número del 1 al 10.

Comandos:

- Variables: `n`, `i`.
- Operadores: `*`.
- Funciones: `input()`, `int()`, `print()`, `range()`.
- Estructuras: `for`.

Ejercicio 8: Números impares usando while

Enunciado:
Muestra los números impares del 1 al 15 usando `while`.

Código:

```
i = 1
while i <= 15:
    if i % 2 != 0:
        print(i)
    i += 1
```

Explicación:

- `while` repite mientras la condición sea verdadera.
- `i += 1` incrementa el contador.

Comandos:

- Variables: `i`.
- Operadores: `%`, `!=`, `<=`, `+=`.
- Funciones: `print()`.
- Estructuras: `while`, `if`.

Ejercicio 9: Suma hasta que el usuario ingrese 0

Enunciado:
Suma números ingresados por el usuario hasta que ingrese 0.

Código:

```
suma = 0
num = int(input("Ingrese un número (0 para terminar): "))
while num != 0:
    suma += num
    num = int(input("Ingrese un número (0 para terminar): "))
print("Suma total:", suma)
```

Explicación:

- `while num != 0` mantiene el bucle hasta ingresar 0.
- Se actualiza `num` en cada iteración.

Comandos:

- Variables: `suma`, `num`.
- Operadores: `!=`, `+=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `while`.

Ejercicio 10: Contar dígitos de un número

Enunciado:
Cuenta cuántos dígitos tiene un número entero positivo.

Código:

```
num = int(input("Ingrese un número positivo: "))
contador = 0
while num > 0:
    num //= 10
    contador += 1
print("Número de dígitos:", contador)
```

Explicación:

- `num //= 10` elimina el último dígito.
- `contador` acumula la cantidad de dígitos.

Comandos:

- Variables: `num`, `contador`.
- Operadores: `//=`, `+=`, `>`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `while`.

Ejercicio 11: Suma de números pares e impares

Enunciado:
Calcula la suma de números pares e impares del 1 al 20 por separado.

Código:

```
suma_pares = 0
suma_impares = 0
for i in range(1, 21):
    if i % 2 == 0:
        suma_pares += i
    else:
        suma_impares += i
print("Suma de pares:", suma_pares)
print("Suma de impares:", suma_impares)
```

Explicación:

- `if i % 2 == 0` identifica pares.
- `else` captura impares.

Comandos:

- Variables: `i`, `suma_pares`, `suma_impares`.
- Operadores: `%`, `==`, `+=`.
- Funciones: `print()`, `range()`.
- Estructuras: `for`, `if`, `else`.

Ejercicio 12: Números primos hasta N

Enunciado:
Muestra todos los números primos hasta un número `N`.

Código:

```
N = int(input("Ingrese N: "))
for num in range(2, N+1):
    primo = True
    for i in range(2, int(num**0.5)+1):
        if num % i == 0:
            primo = False
            break
    if primo:
        print(num, "es primo")
```

Explicación:

- `for i in range(2, int(num**0.5)+1)` busca divisores.
- `break` detiene el bucle si se encuentra un divisor.

Comandos:

- Variables: `N`, `num`, `i`, `primo`.
- Operadores: `%`, `==`, `**`, `+`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `for`, `if`, `break`.

Ejercicio 13: Contar vocales en una cadena

Enunciado:
Cuenta cuántas vocales tiene una palabra ingresada por el usuario.

Código:

```
palabra = input("Ingrese una palabra: ").lower()
contador = 0
for letra in palabra:
    if letra in "aeiou":
        contador += 1
print("Número de vocales:", contador)
```

Explicación:

- `.lower()` convierte la palabra a minúsculas.
- `in "aeiou"` verifica si la letra es vocal.

Comandos:

- Variables: `palabra`, `letra`, `contador`.
- Operadores: `in`, `+=`.
- Funciones: `input()`, `lower()`, `print()`.
- Estructuras: `for`, `if`.

Ejercicio 14: Números múltiplos de 3

Enunciado:
Muestra los números múltiplos de 3 entre 1 y 50.

Código:

```
for i in range(1, 51):
    if i % 3 == 0:
        print(i)
```

Explicación:

- `% 3 == 0` filtra múltiplos de 3.

Comandos:

- Variables: `i`.
- Operadores: `%`, `==`.
- Funciones: `print()`, `range()`.
- Estructuras: `for`, `if`.

Ejercicio 15: Contar caracteres de una cadena

Enunciado:
Cuenta cuántos caracteres tiene una cadena ingresada por el usuario.

Código:

```
texto = input("Ingrese un texto: ")
contador = 0
for c in texto:
    contador += 1
print("Número de caracteres:", contador)
```

Explicación:

- `for c in texto` recorre cada carácter.
- `contador += 1` incrementa el conteo.

Comandos:

- Variables: `texto`, `c`, `contador`.
- Operadores: `+=`.
- Funciones: `input()`, `print()`.
- Estructuras: `for`.

Ejercicio 16: Tabla de multiplicar múltiple

Enunciado:
Muestra las tablas de multiplicar del 1 al 5.

Código:

```
for i in range(1, 6):
    print("Tabla del", i)
    for j in range(1, 11):
        print(i, "x", j, "=", i*j)
    print()
```

Explicación:

- Bucle anidado: `for j` dentro de `for i`.
- `print()` vacío crea espacio entre tablas.

Comandos:

- Variables: `i`, `j`.
- Operadores: `*`.
- Funciones: `print()`, `range()`.
- Estructuras: `for` anidado.

Ejercicio 17: Sumar dígitos de un número

Enunciado:
Suma los dígitos de un número ingresado.

Código:

```
num = int(input("Ingrese un número: "))
suma = 0
while num > 0:
    suma += num % 10
    num //= 10
print("Suma de dígitos:", suma)
```

Explicación:

- `% 10` obtiene el último dígito.
- `// 10` elimina el último dígito.

Comandos:

- Variables: `num`, `suma`.
- Operadores: `%`, `//`, `+=`, `>`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `while`.

Ejercicio 18: Invertir una cadena

Enunciado:
Muestra la cadena ingresada al revés.

Código:

```
texto = input("Ingrese un texto: ")
invertido = ""
for c in texto:
    invertido = c + invertido
print("Texto invertido:", invertido)
```

Explicación:

- Concatenación `c + invertido` construye la cadena invertida.

Comandos:

- Variables: `texto`, `c`, `invertido`.
- Operadores: `+`.
- Funciones: `input()`, `print()`.
- Estructuras: `for`.

Ejercicio 19: Contar letras y números

Enunciado:
Cuenta cuántas letras y números hay en una cadena.

Código:

```
texto = input("Ingrese un texto: ")
letras = 0
numeros = 0
for c in texto:
    if c.isalpha():
        letras += 1
    elif c.isdigit():
        numeros += 1
print("Letras:", letras)
print("Números:", numeros)
```

Explicación:

- `.isalpha()` detecta letras.
- `.isdigit()` detecta dígitos.

Comandos:

- Variables: `texto`, `c`, `letras`, `numeros`.
- Operadores: `+=`.
- Funciones: `input()`, `isalpha()`, `isdigit()`, `print()`.
- Estructuras: `for`, `if`, `elif`.

Ejercicio 20: Patrón de asteriscos

Enunciado:
Muestra un triángulo de asteriscos con 5 filas.

Código:

```
for i in range(1, 6):
    print("*" * i)
```

Explicación:

- `"*"` se repite `i` veces para formar el patrón.

Comandos:

- Variables: `i`.
- Operadores: `*` (repetición de cadenas).
- Funciones: `print()`, `range()`.
- Estructuras: `for`.

Ejercicio 21: Suma de números ingresados en lista

Enunciado:
Pide al usuario 5 números, guárdalos en una lista y muestra la suma total.

Código:

```
numeros = []
for i in range(5):
    n = int(input(f"Ingrese el número {i+1}: "))
    numeros.append(n)
suma = 0
for num in numeros:
    suma += num
print("Suma total:", suma)
```

Explicación:

- `append()` añade elementos a la lista.
- Segundo bucle recorre la lista para sumar.

Comandos:

- Variables: `numeros`, `i`, `n`, `num`, `suma`.
- Operadores: `+=`.
- Funciones: `input()`, `int()`, `print()`, `append()`, `range()`.
- Estructuras: `for`.

Ejercicio 22: Encontrar el mayor de una lista

Enunciado:
Pide 5 números, guárdalos en una lista y muestra el mayor.

Código:

```
numeros = []
for i in range(5):
    n = int(input(f"Ingrese el número {i+1}: "))
    numeros.append(n)
mayor = numeros[0]
for num in numeros:
    if num > mayor:
        mayor = num
print("El número mayor es:", mayor)
```

Explicación:

- Inicializamos `mayor` con el primer elemento.
- `if num > mayor` actualiza el mayor.

Comandos:

- Variables: `numeros`, `i`, `n`, `num`, `mayor`.
- Operadores: `>`.
- Funciones: `input()`, `int()`, `print()`, `append()`, `range()`.
- Estructuras: `for`, `if`.

Ejercicio 23: Promedio de lista

Enunciado:
Pide 5 números, calcula y muestra el promedio.

Código:

```
numeros = []
for i in range(5):
    n = float(input(f"Ingrese el número {i+1}: "))
    numeros.append(n)
suma = 0
for num in numeros:
    suma += num
promedio = suma / len(numeros)
print("Promedio:", promedio)
```

Explicación:

- `len(numeros)` devuelve cantidad de elementos.
- Promedio = suma / cantidad de elementos.

Comandos:

- Variables: `numeros`, `i`, `n`, `num`, `suma`, `promedio`.
- Operadores: `+=`, `/`.
- Funciones: `input()`, `float()`, `print()`, `append()`, `len()`, `range()`.
- Estructuras: `for`.

Ejercicio 24: Contar números positivos y negativos

Enunciado:
Pide 10 números y cuenta cuántos son positivos y cuántos negativos.

Código:

```
positivos = 0
negativos = 0
for i in range(10):
    n = int(input(f"Ingrese el número {i+1}: "))
    if n > 0:
        positivos += 1
    elif n < 0:
        negativos += 1
print("Positivos:", positivos)
print("Negativos:", negativos)
```

Explicación:

- `if` y `elif` clasifican los números.

Comandos:

- Variables: `positivos`, `negativos`, `i`, `n`.
- Operadores: `>`, `<`, `+=`.
- Funciones: `input()`, `int()`, `print()`, `range()`.
- Estructuras: `for`, `if`, `elif`.

Ejercicio 25: Suma de números pares de lista

Enunciado:
Pide 10 números, guárdalos en una lista y suma solo los pares.

Código:

```
numeros = []
for i in range(10):
    n = int(input(f"Ingrese el número {i+1}: "))
    numeros.append(n)
suma_pares = 0
for num in numeros:
    if num % 2 == 0:
        suma_pares += num
print("Suma de pares:", suma_pares)
```

Explicación:

- `% 2 == 0` identifica los pares.
- Solo se suman los números pares.

Comandos:

- Variables: `numeros`, `i`, `n`, `num`, `suma_pares`.
- Operadores: `%`, `==`, `+=`.
- Funciones: `input()`, `int()`, `print()`, `append()`, `range()`.
- Estructuras: `for`, `if`.

Ejercicio 26: Invertir lista

Enunciado:
Pide 5 números, guárdalos en una lista y muestra la lista invertida.

Código:

```
numeros = []
for i in range(5):
    n = int(input(f"Ingrese el número {i+1}: "))
    numeros.append(n)
invertida = []
for i in range(len(numeros)-1, -1, -1):
    invertida.append(numeros[i])
print("Lista invertida:", invertida)
```

Explicación:

- `range(len(numeros)-1, -1, -1)` recorre la lista de atrás hacia adelante.

Comandos:

- Variables: `numeros`, `invertida`, `i`, `n`.
- Operadores: `-`, `append()`.
- Funciones: `input()`, `int()`, `print()`, `len()`, `range()`.
- Estructuras: `for`.

Ejercicio 27: Contar vocales en lista de palabras

Enunciado:
Pide 3 palabras, cuenta las vocales de cada palabra.

Código:

```
palabras = []
for i in range(3):
    palabra = input(f"Ingrese la palabra {i+1}: ").lower()
    palabras.append(palabra)

for palabra in palabras:
    contador = 0
    for letra in palabra:
        if letra in "aeiou":
            contador += 1
    print(f"Vocales en {palabra}:", contador)
```

Explicación:

- Bucle anidado: recorre cada palabra y sus letras.
- `in "aeiou"` verifica vocal.

Comandos:

- Variables: `palabras`, `i`, `palabra`, `letra`, `contador`.
- Operadores: `in`, `+=`.
- Funciones: `input()`, `lower()`, `print()`, `append()`, `range()`.
- Estructuras: `for`, `if`.

Ejercicio 28: Números pares e impares en lista

Enunciado:
Pide 10 números y muestra cuántos son pares y cuántos impares.

Código:

```
numeros = []
for i in range(10):
    n = int(input(f"Ingrese el número {i+1}: "))
    numeros.append(n)
pares = 0
impares = 0
for num in numeros:
    if num % 2 == 0:
        pares += 1
    else:
        impares += 1
print("Pares:", pares)
print("Impares:", impares)
```

Explicación:

- Similar al ejercicio 24, pero usando lista para almacenar los números.

Comandos:

- Variables: `numeros`, `i`, `n`, `num`, `pares`, `impares`.
- Operadores: `%`, `==`, `+=`.
- Funciones: `input()`, `int()`, `print()`, `append()`, `range()`.
- Estructuras: `for`, `if`, `else`.

Ejercicio 29: Suma de elementos positivos de lista

Enunciado:
Pide 8 números, guarda en lista y suma solo los positivos.

Código:

```
numeros = []
for i in range(8):
    n = int(input(f"Ingrese el número {i+1}: "))
    numeros.append(n)
suma_positivos = 0
for num in numeros:
    if num > 0:
        suma_positivos += num
print("Suma de positivos:", suma_positivos)
```

Explicación:

- `if num > 0` filtra solo los números positivos.

Comandos:

- Variables: `numeros`, `i`, `n`, `num`, `suma_positivos`.
- Operadores: `>`, `+=`.
- Funciones: `input()`, `int()`, `print()`, `append()`, `range()`.
- Estructuras: `for`, `if`.

Ejercicio 30: Patrón de asteriscos cuadrado

Enunciado:
Muestra un cuadrado de asteriscos de 5x5 usando bucles anidados.

Código:

```
for i in range(5):
    for j in range(5):
        print("*", end=" ")
    print()
```

Explicación:

- Bucle anidado: `i` controla filas, `j` columnas.
- `end=" "` evita salto de línea en cada `print` de columna.

Comandos:

- Variables: `i`, `j`.
- Operadores: N/A.
- Funciones: `print()`, `range()`.
- Estructuras: `for` anidado.

Ejercicio 31: Validar entrada positiva

Enunciado:
Pide al usuario un número positivo; si ingresa un número negativo, sigue pidiendo hasta que sea positivo.

Código:

```
num = int(input("Ingrese un número positivo: "))
while num < 0:
    print("Número inválido, intente de nuevo.")
    num = int(input("Ingrese un número positivo: "))
print("Número aceptado:", num)
```

Explicación:

- `while num < 0` repite mientras el número sea negativo.
- Se actualiza `num` en cada iteración.

Comandos:

- Variables: `num`.
- Operadores: `<`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `while`.

Ejercicio 32: Sumar números hasta que se ingrese 0

Enunciado:
Suma todos los números ingresados por el usuario hasta que ingrese 0.

Código:

```
suma = 0
while True:
    num = int(input("Ingrese un número (0 para terminar): "))
    if num == 0:
        break
    suma += num
print("Suma total:", suma)
```

Explicación:

- `while True` crea un bucle infinito.
- `break` detiene el bucle cuando `num == 0`.

Comandos:

- Variables: `suma`, `num`.
- Operadores: `==`, `+=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `while`, `if`, `break`.

Ejercicio 33: Factorial con while

Enunciado:
Calcula el factorial de un número usando `while`.

Código:

```
n = int(input("Ingrese un número: "))
factorial = 1
i = 1
while i <= n:
    factorial *= i
    i += 1
print("Factorial de", n, "es", factorial)
```

Explicación:

- `i` incrementa hasta `n`.
- `factorial *= i` acumula el producto.

Comandos:

- Variables: `n`, `factorial`, `i`.
- Operadores: `<=`, `*=`, `+=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `while`.

Ejercicio 34: Contar dígitos pares e impares

Enunciado:
Cuenta cuántos dígitos pares e impares tiene un número entero positivo.

Código:

```
num = int(input("Ingrese un número: "))
pares = 0
impares = 0
while num > 0:
    digito = num % 10
    if digito % 2 == 0:
        pares += 1
    else:
        impares += 1
    num //= 10
print("Dígitos pares:", pares)
print("Dígitos impares:", impares)
```

Explicación:

- `% 10` obtiene el último dígito.
- `// 10` elimina el último dígito.

Comandos:

- Variables: `num`, `digito`, `pares`, `impares`.
- Operadores: `%`, `//`, `==`, `+=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `while`, `if`, `else`.

Ejercicio 35: Patrón pirámide

Enunciado:
Muestra una pirámide de asteriscos de 5 filas.

Código:

```
for i in range(1, 6):
    print(" " * (5-i) + "*" * (2*i-1))
```

Explicación:

- `" " * (5-i)` crea espacios iniciales.
- `"*" * (2*i-1)` genera los asteriscos de la fila.

Comandos:

- Variables: `i`.
- Operadores: `*`, `-`.
- Funciones: `print()`, `range()`.
- Estructuras: `for`.

Ejercicio 36: Número primo usando while

Enunciado:
Verifica si un número ingresado es primo usando `while`.

Código:

```
num = int(input("Ingrese un número: "))
if num < 2:
    print("No es primo")
else:
    i = 2
    primo = True
    while i <= num // 2:
        if num % i == 0:
            primo = False
            break
        i += 1
    if primo:
        print("Es primo")
    else:
        print("No es primo")
```

Explicación:

- `while i <= num // 2` revisa divisores hasta la mitad.
- `break` detiene la búsqueda si se encuentra un divisor.

Comandos:

- Variables: `num`, `i`, `primo`.
- Operadores: `%`, `<=`, `==`, `+=`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `if`, `while`, `break`.

Ejercicio 37: Contar ocurrencias de una letra

Enunciado:

Cuenta cuántas veces aparece una letra en una palabra.

Código:

```
palabra = input("Ingrese una palabra: ").lower()
letra_buscar = input("Ingrese la letra a buscar: ").lower()
contador = 0
for letra in palabra:
    if letra == letra_buscar:
        contador += 1
print(f"La letra '{letra_buscar}' aparece {contador} veces")
```

Explicación:

- `.lower()` normaliza la entrada.
- `==` compara cada letra con la buscada.

Comandos:

- Variables: `palabra`, `letra_buscar`, `letra`, `contador`.
- Operadores: `==`, `+=`.
- Funciones: `input()`, `lower()`, `print()`.
- Estructuras: `for`, `if`.

Ejercicio 38: Patrón cuadrado con diagonales

Enunciado:
Muestra un cuadrado 5x5 de asteriscos con diagonales vacías.

Código:

```
for i in range(5):
    for j in range(5):
        if i == 0 or i == 4 or j == 0 or j == 4 or i == j or i + j == 4:
            print("*", end="")
        else:
            print(" ", end="")
    print()
```

Explicación:

- Bucle anidado recorre filas y columnas.
- Condiciones combinadas crean bordes y diagonales.

Comandos:

- Variables: `i`, `j`.
- Operadores: `==`, `+`.
- Funciones: `print()`, `range()`.
- Estructuras: `for` anidado, `if`, `else`.

Ejercicio 39: Promedio con validación

Enunciado:
Pide al usuario números hasta que ingrese 0, y muestra el promedio de los ingresados (excepto 0).

Código:

```
suma = 0
contador = 0
while True:
    n = int(input("Ingrese un número (0 para terminar): "))
    if n == 0:
        break
    suma += n
    contador += 1
if contador > 0:
    promedio = suma / contador
    print("Promedio:", promedio)
else:
    print("No se ingresaron números")
```

Explicación:

- `contador` evita división por cero.
- `while True` con `break` controla finalización.

Comandos:

- Variables: `suma`, `contador`, `n`, `promedio`.
- Operadores: `+=`, `/`, `==`, `>`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `while`, `if`, `break`, `else`.

Ejercicio 40: Patrón de triángulo invertido

Enunciado:
Muestra un triángulo invertido de asteriscos de 5 filas.

Código:

```
for i in range(5, 0, -1):
    print("*" * i)
```

Explicación:

- `range(5, 0, -1)` recorre de 5 a 1.
- `"*"` se repite `i` veces por fila.

Comandos:

- Variables: `i`.
- Operadores: `*`, `-`.
- Funciones: `print()`, `range()`.
- Estructuras: `for`.

Ejercicio 41: Patrón de pirámide invertida

Enunciado:
Muestra una pirámide invertida de asteriscos de 5 filas.

Código:

```
for i in range(5, 0, -1):
    print(" " * (5 - i) + "*" * (2*i - 1))
```

Explicación:

- `" " * (5 - i)` genera los espacios iniciales.
- `"*" * (2*i - 1)` forma la fila de asteriscos.

Comandos:

- Variables: `i`.
- Operadores: `*`, `-`.
- Funciones: `print()`, `range()`.
- Estructuras: `for`.

Ejercicio 42: Suma de múltiplos de 3 y 5

Enunciado:
Calcula la suma de los números del 1 al 100 que son múltiplos de 3 o 5.

Código:

```
suma = 0
for i in range(1, 101):
    if i % 3 == 0 or i % 5 == 0:
        suma += i
print("Suma de múltiplos de 3 o 5:", suma)
```

Explicación:

- `or` permite sumar números que cumplan cualquiera de las condiciones.

Comandos:

- Variables: `suma`, `i`.
- Operadores: `%`, `==`, `or`, `+=`.
- Funciones: `print()`, `range()`.
- Estructuras: `for`, `if`.

Ejercicio 43: Validación de contraseña

Enunciado:
Pide una contraseña que cumpla con longitud >= 6 y contenga un número; sigue pidiendo hasta que sea válida.

Código:

```
while True:
    contraseña = input("Ingrese una contraseña: ")
    if len(contraseña) >= 6 and any(c.isdigit() for c in contraseña):
        break
    print("Contraseña inválida, intente de nuevo.")
print("Contraseña aceptada")
```

Explicación:

- `len(contraseña) >= 6` valida longitud.
- `any(c.isdigit() for c in contraseña)` verifica al menos un dígito.

Comandos:

- Variables: `contraseña`, `c`.
- Operadores: `>=`, `and`.
- Funciones: `input()`, `len()`, `print()`, `any()`, `isdigit()`.
- Estructuras: `while`, `if`, `break`.

Ejercicio 44: Patrón de números

Enunciado:

Muestra el siguiente patrón:

```
1
12
123
1234
12345
```

Código:

```
for i in range(1, 6):
    for j in range(1, i+1):
        print(j, end="")
    print()
```

Explicación:

- Bucle anidado: `i` filas, `j` números por fila.
- `end=""` evita salto de línea en la fila.

Comandos:

- Variables: `i`, `j`.
- Operadores: `+`.
- Funciones: `print()`, `range()`.
- Estructuras: `for` anidado.

Ejercicio 45: Promedio de números positivos

Enunciado:
Pide números hasta que se ingrese 0 y muestra el promedio de los positivos.

Código:

```
suma = 0
contador = 0
while True:
    n = int(input("Ingrese un número (0 para terminar): "))
    if n == 0:
        break
    if n > 0:
        suma += n
        contador += 1
if contador > 0:
    print("Promedio de positivos:", suma / contador)
else:
    print("No se ingresaron números positivos")
```

Explicación:

- Solo se suman los números positivos.
- Validación evita división por cero.

Comandos:

- Variables: `suma`, `contador`, `n`.
- Operadores: `>`, `==`, `+=`, `/`.
- Funciones: `input()`, `int()`, `print()`.
- Estructuras: `while`, `if`, `break`.

Ejercicio 46: Tabla de multiplicar completa

Enunciado:
Muestra las tablas de multiplicar del 1 al 10.

Código:

```
for i in range(1, 11):
    for j in range(1, 11):
        print(f"{i} x {j} = {i*j}")
    print()
```

Explicación:

- Bucle anidado: `i` tablas, `j` multiplicadores.
- `f-string` formatea la salida.

Comandos:

- Variables: `i`, `j`.
- Operadores: `*`.
- Funciones: `print()`, `range()`.
- Estructuras: `for` anidado.

Ejercicio 47: Patrón de diamante

Enunciado:
Muestra un diamante de asteriscos de 5 filas por lado.

Código:

```
n = 5
for i in range(1, n+1):
    print(" "*(n-i) + "*"*(2*i-1))
for i in range(n-1, 0, -1):
    print(" "*(n-i) + "*"*(2*i-1))
```

Explicación:

- Primer bucle: pirámide superior.
- Segundo bucle: pirámide inferior invertida.

Comandos:

- Variables: `i`, `n`.
- Operadores: `*`, `-`.
- Funciones: `print()`, `range()`.
- Estructuras: `for`.

Ejercicio 48: Contar letras y dígitos en texto

Enunciado:
Cuenta cuántas letras y dígitos hay en un texto ingresado.

Código:

```
texto = input("Ingrese un texto: ")
letras = 0
digitos = 0
for c in texto:
    if c.isalpha():
        letras += 1
    elif c.isdigit():
        digitos += 1
print("Letras:", letras)
print("Dígitos:", digitos)
```

Explicación:

- `.isalpha()` identifica letras.
- `.isdigit()` identifica números.

Comandos:

- Variables: `texto`, `c`, `letras`, `digitos`.
- Operadores: `+=`.
- Funciones: `input()`, `isalpha()`, `isdigit()`, `print()`.
- Estructuras: `for`, `if`, `elif`.

Ejercicio 49: Suma de diagonales en matriz

Enunciado:
Dada una matriz 3x3 de números ingresados por el usuario, suma las diagonales principal y secundaria.

Código:

```
matriz = []
for i in range(3):
    fila = []
    for j in range(3):
        n = int(input(f"Ingrese elemento [{i}][{j}]: "))
        fila.append(n)
    matriz.append(fila)

suma_principal = 0
suma_secundaria = 0
for i in range(3):
    suma_principal += matriz[i][i]
    suma_secundaria += matriz[i][2-i]

print("Suma diagonal principal:", suma_principal)
print("Suma diagonal secundaria:", suma_secundaria)
```

Explicación:

- Bucle anidado para ingresar la matriz.
- `matriz[i][i]` diagonal principal.
- `matriz[i][2-i]` diagonal secundaria.

Comandos:

- Variables: `matriz`, `fila`, `i`, `j`, `n`, `suma_principal`, `suma_secundaria`.
- Operadores: `+=`.
- Funciones: `input()`, `int()`, `print()`, `append()`, `range()`.
- Estructuras: `for` anidado.

Ejercicio 50: Patrón de números en espiral

Enunciado:
Muestra un patrón 4x4 de números del 1 al 16 en forma de espiral (simulado con bucles simples).

Código:

```
numeros = list(range(1, 17))
for i in range(4):
    for j in range(4):
        print(f"{numeros[i*4+j]:2}", end=" ")
    print()
```

Explicación:

- `numeros[i*4+j]` selecciona elementos por fila.
- `:2` en `f-string` alinea los números.

Comandos:

- Variables: `numeros`, `i`, `j`.
- Operadores: `*`, `+`.
- Funciones: `list()`, `range()`, `print()`.
- Estructuras: `for` anidado.

MANUAL DETALLADO DE EJERCICIOS DE PYTHON: ESTRUCTURAS DE DATOS (LISTAS)

Ejercicios 1 a 10: Operaciones Fundamentales con Listas

N°	Enunciado y Tarea	Código de Implementación	Explicación Detallada	Controles y Comandos
1	**Crear una lista y mostrarla:** Crea una lista con los números del 1 al 5 y muéstrala.	`numeros = print("Lista de números:", numeros)`	La lista se crea utilizando **corchetes `[]`**. La función `print()` muestra el contenido completo de la lista en la consola.	**Variables:** `numeros`. **Funciones:** `print()`.
2	**Acceder a elementos de la lista:** Muestra el primer y último elemento de la lista ``.	`numeros = print("Primer elemento:", numeros) print("Último elemento:", numeros[-1])`	Se utiliza el **índice** `` para acceder al primer elemento, ya que la numeración empieza en 0. Se utiliza el **índice `[-1]`** para acceder al último elemento.	**Variables:** `numeros`. **Operadores:** Índices `[ ]`. **Funciones:** `print()`.
3	**Modificar elementos:** Cambia el tercer elemento de `` a 10 y muestra la lista.	`numeros = numeros = 10 print("Lista modificada:", numeros)`	**`numeros = 10`** reemplaza el valor existente en el índice 2 (el tercer elemento) por el nuevo valor (10).	**Variables:** `numeros`. **Operadores:** Índices `[ ]`, `=` (asignación). **Funciones:** `print()`.
4	**Agregar elementos:** Agrega el número 6 al final de la lista ``.	`numeros = numeros.append(6) print("Lista después de append:", numeros)`	La función **`append()`** añade un elemento *al final* de la lista.	**Variables:** `numeros`. **Funciones:** `append()`, `print()`.
5	**Insertar elementos:** Inserta el número 0 al inicio de la lista ``.	`numeros = numeros.insert(0, 0) print("Lista después de insert:", numeros)`	La función **`insert(index, valor)`** inserta un elemento (`valor`) en la posición especificada (`index`). En este caso, el índice 0 es el inicio.	**Variables:** `numeros`. **Funciones:** `insert()`, `print()`.
6	**Eliminar elementos por valor:** Elimina el número 3 de la lista ``.	`numeros = numeros.remove(3) print("Lista después de remove:", numeros)`	La función **`remove(valor)`** busca y elimina la primera coincidencia del valor especificado.	**Variables:** `numeros`. **Funciones:** `remove()`, `print()`.
7	**Eliminar elemento por índice:** Elimina el tercer	`numeros = del numeros print("Lista después de del:", numeros)`	El comando **`del`** elimina un elemento de la lista utilizando su posición (índice). El	**Variables:** `numeros`. **Operadores:** `del`, índices `[ ]`.

	elemento de `` usando su índice.		tercer elemento se encuentra en el índice 2.	**Funciones:** `print()`.
8	**Contar elementos:** Cuenta cuántas veces aparece el número 2 en ``.	`numeros = cantidad = numeros.count(2) print("Número de veces que aparece 2:", cantidad)`	La función **`count(valor)`** devuelve la cantidad de veces que el `valor` aparece dentro de la lista.	**Variables:** `numeros`, `cantidad`. **Funciones:** `count()`, `print()`.
9	**Ordenar lista:** Ordena de menor a mayor la lista ``.	`numeros = numeros.sort() print("Lista ordenada:", numeros)`	La función **`sort()`** ordena los elementos de la lista *in place* (modifica la lista original) en forma ascendente (menor a mayor).	**Variables:** `numeros`. **Funciones:** `sort()`, `print()`.
10	**Lista inversa:** Invierte el orden de la lista ``.	`numeros = numeros.reverse() print("Lista invertida:", numeros)`	La función **`reverse()`** invierte el orden de los elementos de la lista, modificándola directamente.	**Variables:** `numeros`. **Funciones:** `reverse()`, `print()`.

Ejercicios 11 a 20: Listas Intermedias (Cálculos y Búsquedas)

N°	Enunciado y Tarea	Código de Implementación	Explicación Detallada	Controles y Comandos
11	**Suma de elementos de la lista:** Calcula la suma de todos los elementos de la lista ``.	`numeros = suma = 0 for num in numeros: suma += num print("Suma total:", suma)`	Se utiliza un bucle **`for`** para recorrer cada elemento (`num`) de la lista y el operador **`+=`** lo suma al acumulador `suma`.	**Estructuras:** `for`. **Operadores:** `+=`. **Funciones:** `print()`.
12	**Promedio de elementos:** Calcula el promedio de los elementos de ``.	*[Se calcula la suma usando un bucle for (ver E11)]* `promedio = suma / len(numeros) print("Promedio:", promedio)`	El promedio se obtiene dividiendo la `suma` entre la cantidad de elementos, la cual se calcula usando la función **`len(numeros)`**.	**Estructuras:** `for`. **Operadores:** `+=`, `/`. **Funciones:** `len()`, `print()`.
13	**Encontrar el máximo:** Encuentra el mayor número en ``.	`mayor = numeros for num in numeros: if num > mayor: mayor = num print("Número mayor:", mayor)`	Se inicializa la variable `mayor` con el primer elemento. El bucle recorre la lista, y la estructura **`if`** compara cada elemento con el valor actual de `mayor` y lo actualiza si se encuentra un número mayor.	**Estructuras:** `for`, `if`. **Operadores:** `>`.
14	**Encontrar el**	`menor = numeros for`	Es similar al ejercicio 13,	**Estructuras:**

	mínimo: Encuentra el menor número en ``.	`num in numeros: if num < menor: menor = num print("Número menor:", menor)`	pero la condición de comparación usa el operador **`<`**.	`for`, `if`. **Operadores:** `<`.
15	**Contar elementos mayores que un valor:** Cuenta cuántos números en `` son mayores que 5.	`contador = 0 for num in numeros: if num > 5: contador += 1 print("Cantidad de números mayores a 5:", contador)`	El **`contador`** se incrementa con `+= 1` únicamente si el número (`num`) satisface la condición **`if num > 5`**.	**Estructuras:** `for`, `if`. **Operadores:** `>`, `+=`.
16	**Lista de números pares:** Crea una nueva lista solo con los números pares de ``.	`pares = [] for num in numeros: if num % 2 == 0: pares.append(num) print("Números pares:", pares)`	La condición **`% 2 == 0`** se utiliza para identificar los números pares (resto de la división es cero). Los pares se añaden a la nueva lista usando **`append()`**.	**Estructuras:** `for`, `if`. **Operadores:** `%`, `==`. **Funciones:** `append()`.
17	**Sumar elementos pares e impares por separado:** Suma los números pares e impares de ``.	`suma_pares = 0 suma_impares = 0 for num in numeros: if num % 2 == 0: suma_pares += num else: suma_impares += num`	Se utilizan dos acumuladores. La estructura **`if-else`** separa los números: si es par, suma a `suma_pares`; si es impar (el `else`), suma a `suma_impares`.	**Estructuras:** `for`, `if`, `else`. **Operadores:** `%`, `==`, `+=`.
18	**Buscar un elemento en la lista:** Verifica si el número 4 está en ``.	`if 4 in numeros: print("El número 4 está en la lista") else: print("El número 4 no está en la lista")`	El operador **`in`** verifica de manera eficiente si un valor existe dentro de la lista. La respuesta se gestiona con una estructura **`if-else`**.	**Estructuras:** `if-else`. **Operadores:** `in`.
19	**Contar ocurrencias de un elemento:** Cuenta cuántas veces aparece el número 2 en ``.	`cantidad = numeros.count(2) print("El número 2 aparece:", cantidad, "veces")`	Se utiliza la función **`count()`** para devolver la cantidad de veces que el valor especificado (2) aparece en la lista.	**Variables:** `cantidad`. **Funciones:** `count()`, `print()`.
20	**Copiar una lista:** Copia la lista `` a otra lista.	`numeros = copia = numeros.copy() print("Copia de la lista:", copia)`	La función **`copy()`** crea una copia *independiente* de la lista original, evitando que las modificaciones en una afecten a la otra.	**Variables:** `numeros`, `copia`. **Funciones:** `copy()`, `print()`.

Ejercicios 21 a 30: Manipulación y Operadores de Listas

N°	Enunciado y Tarea	Código de Implementación	Explicación Detallada	Controles y Comandos
21	**Invertir una**	`numeros.reverse()`	La función	**Funciones:**

	lista: Invierte la lista `para que quede`.	`print("Lista invertida:", numeros)`	**`reverse()`** invierte el orden de los elementos en su lugar (modificando la lista original).	`reverse()`, `print()`.
22	**Ordenar lista de menor a mayor:** Ordena la lista `` en orden ascendente.	`numeros.sort() print("Lista ordenada:", numeros)`	La función **`sort()`** ordena los elementos en orden ascendente (de menor a mayor).	**Funciones:** `sort()`, `print()`.
23	**Ordenar lista de mayor a menor:** Ordena la lista `` en orden descendente.	`numeros.sort(reverse=True) print("Lista descendente:", numeros)`	Se utiliza la función **`sort()`** pero se añade el argumento **`reverse=True`** para forzar el orden de mayor a menor (descendente).	**Funciones:** `sort()`, `print()`.
24	**Eliminar duplicados:** Elimina los números repetidos de ``.	`sin_duplicados = list(set(numeros)) print("Lista sin duplicados:", sin_duplicados)`	Se convierte la lista a un **`set()`** (conjunto), que por definición elimina elementos repetidos. Luego se convierte el conjunto de nuevo a una **`list()`**.	**Funciones:** `set()`, `list()`, `print()`.
25	**Concatenar dos listas:** Une las listas `y`.	`lista_total = lista1 + lista2 print("Lista concatenada:", lista_total)`	El operador **+** (suma) se utiliza para concatenar o unir dos listas en una nueva lista.	**Operadores:** `+`. **Variables:** `lista_total`.
26	**Multiplicar lista:** Crea una lista que repita `` tres veces.	`repetida = numeros * 3 print("Lista repetida:", repetida)`	El operador de multiplicación **`*`** repite la secuencia de la lista tantas veces como se indique (en este caso, 3).	**Operadores:** `*`.
27	**Sumar dos listas elemento por elemento:** Suma `con` para obtener ``.	`resultado = [] for i in range(len(lista1)): resultado.append(lista1[i] + lista2[i]) print("Suma elemento a elemento:", resultado)`	Se recorren los **índices** con `range(len(lista1))`. Se suman los elementos correspondientes de ambas listas en la misma posición `lista1[i] + lista2[i]`.	**Estructuras:** `for`. **Funciones:** `len()`, `range()`, `append()`. **Operadores:** `+`.
28	**Lista con cuadrados de números:** Crea	`for num in numeros: cuadrados.append(num ** 2)`	Dentro del bucle **`for`**, el operador **`** 2`** calcula el cuadrado	**Estructuras:** `for`. **Operadores:** `**`. **Funciones:**

	una lista con los cuadrados de ``.	`print("Cuadrados:", cuadrados)`	(elevación a la segunda potencia) de cada número.	`append()`.
29	**Intercambiar dos elementos:** Intercambia el primer y último elemento de ``.	`numeros, numeros[-1] = numeros[-1], numeros print("Lista modificada:", numeros)`	Se utiliza la **asignación múltiple** (`a, b = b, a`) para intercambiar los valores del primer elemento (`numeros`) y el último elemento (`numeros[-1]`).	**Operadores:** asignación múltiple.
30	**Lista con números al revés usando slicing:** Invierte `` usando *slicing*.	`invertida = numeros[::-1] print("Lista invertida con slicing:", invertida)`	El operador de **slicing** (rebanado) `[::-1]` crea una nueva lista donde se recorre la original en orden inverso.	**Operadores:** slicing (`[inicio:fin:paso]`).

Ejercicios 31 a 40: Listas Anidadas (Matrices)

N°	Enunciado y Tarea	Código de Implementación	Explicación Detallada	Controles y Comandos
31	**Crear una matriz 3x3:** Crea una matriz de 3x3 con los números del 1 al 9.	`matriz = [,,] for fila in matriz: print(fila)`	Una matriz se representa en Python como una **lista de listas** (una lista que contiene otras listas, donde cada sublista es una fila).	**Estructuras:** lista de listas, `for`.
32	**Acceder a un elemento de la matriz:** Muestra el número en la segunda fila, tercera columna.	`print("Elemento en fila 2, columna 3:", matriz)`	Se utilizan **índices dobles `[i][j]`**. `matriz` accede a la segunda fila (índice 1), y `` accede al tercer elemento de esa fila (índice 2).	**Operadores:** índices dobles `[i][j]`.
33	**Recorrer todos los elementos de la matriz:** Muestra cada número de la matriz 3x3 por separado.	`for fila in matriz: for elemento in fila: print(elemento, end=" ")`	Se usan **bucles `for` anidados**: el primero recorre las filas, y el segundo recorre cada `elemento` dentro de cada fila. El parámetro `end=" "` se usa en `print()` para mantener los elementos en la misma línea.	**Estructuras:** `for` anidados.
34	**Sumar todos los elementos de la matriz:** Suma todos los elementos de la	`suma = 0 for fila in matriz: for elemento in fila: suma += elemento print("Suma total:", suma)`	Se utilizan bucles `for` anidados para acceder a cada `elemento`, y se acumula la suma con el operador `+=`.	**Operadores:** `+=`. **Estructuras:** `for` anidado.

	matriz 3x3.			
35	**Crear una lista con los números pares de la matriz:** Guarda los pares de la matriz.	`pares = []` *[for anidados]* `if elemento % 2 == 0: pares.append(elemento) print("Números pares:", pares)`	Se verifica si cada elemento es par usando la condición **`elemento % 2 == 0`**. Si se cumple, el elemento se añade a la lista `pares` usando `append()`.	**Estructuras:** `for`, `if`. **Operadores:** `%`, `==`.
36	**Transponer una matriz:** Transforma la matriz 3x3 en su transpuesta (filas a columnas).	`transpuesta = [] for i in range(len(matriz)): for j in range(len(matriz)): fila.append(matriz[j][i])`	Se utilizan bucles anidados que acceden a los elementos mediante `matriz[j][i]`, **intercambiando los índices** de fila (`j`) y columna (`i`) para lograr la transposición.	**Estructuras:** `for` anidados. **Funciones:** `len()`, `append()`.
37	**Generar lista con los cuadrados de una matriz:** Crea una nueva matriz con los cuadrados de cada número.	*[for anidados]* `nueva_fila.append(elemento ** 2) cuadrados.append(nueva_fila)`	Se recorre la matriz con bucles anidados. Cada `elemento` se eleva al cuadrado mediante el operador **`** 2`**.	**Estructuras:** `for` anidados. **Operadores:** `**`.
38	**Sumar dos matrices:** Suma dos matrices elemento por elemento.	*[for anidados]* `fila.append(A[i][j] + B[i][j]) suma.append(fila)`	Se utilizan bucles anidados para recorrer la matriz por índices `i` (fila) y `j` (columna). Se suman los elementos que ocupan la **misma posición `[i][j]`** en ambas matrices (`A` y `B`).	**Estructuras:** `for` anidados. **Funciones:** `len()`, `append()`.
39	**Buscar un número en la matriz:** Verifica si el número 5 está en la matriz.	`encontrado = False for fila in matriz: if 5 in fila: encontrado = True break`	Se recorren las filas. Se utiliza el operador **`in`** para verificar si el número está dentro de la fila actual. Si se encuentra, la instrucción **`break`** detiene el bucle para ahorrar procesamiento.	**Estructuras:** `for`, `if`, `break`. **Operadores:** `in`.
40	**Calcular la diagonal principal de una matriz:** Muestra los elementos de la diagonal principal de la matriz 3x3.	`diagonal = [] for i in range(len(matriz)): diagonal.append(matriz[i][i]) print("Diagonal principal:", diagonal)`	La diagonal principal se obtiene accediendo a los elementos donde el índice de fila y columna son iguales (`,,`), utilizando solo un índice `i`.	**Estructuras:** `for`. **Funciones:** `len()`, `append()`.

Ejercicios 41 a 50: Comprensión de Listas y Funciones Avanzadas

N°	Enunciado y Tarea	Código de Implementación	Explicación Detallada	Controles y Comandos
41	**Crear una lista de números del 1 al 10 con comprensión:** Genera automáticamente la lista `[1, 2, 3, ..., 10]`.	`numeros = [i for i in range(1, 11)] print("Lista generada:", numeros)`	Se usa la **comprensión de listas** `[i for i in ...]`. La función **`range(1, 11)`** genera una secuencia que va desde 1 hasta 10.	**Funciones:** `range()`, `print()`. **Operadores:** `for` dentro de lista.
42	**Lista de cuadrados con comprensión:** Crea una lista con los cuadrados de ``.	`cuadrados = [n**2 for n in numeros] print("Cuadrados:", cuadrados)`	Dentro de la comprensión, se aplica el cálculo **`n**2`** directamente a cada elemento (`n`) de la lista de números.	**Operadores:** `**`.
43	**Filtrar números pares con comprensión:** Crea una lista con solo los números pares de ``.	`pares = [n for n in numeros if n % 2 == 0] print("Números pares:", pares)`	Se añade una condición **`if n % 2 == 0`** al final de la comprensión para filtrar solo aquellos elementos que cumplen la condición de ser pares.	**Estructuras:** `if` en comprensión. **Operadores:** `%`, `==`.
44	**Elevar a la tercera potencia solo los impares:** Crea una lista con el cubo de los números impares en ``.	`cubos_impares = [n**3 for n in numeros if n % 2 != 0] print("Cubos de impares:", cubos_impares)`	Se calcula el cubo (**`n**3`**) para cada elemento que satisface la condición **`if n % 2 != 0`** (es decir, el resto no es cero, por lo tanto, es impar).	**Operadores:** `%`, `!=`, `**`.
45	**Usar max y min en listas:** Encuentra el mayor y menor número de ``.	`print("Máximo:", max(numeros)) print("Mínimo:", min(numeros))`	La función integrada **`max(lista)`** devuelve el elemento mayor, y **`min(lista)`** devuelve el elemento menor.	**Funciones:** `max()`, `min()`, `print()`.
46	**Usar sum en listas:** Calcula la suma total de ``.	`print("Suma total:", sum(numeros))`	La función integrada **`sum(lista)`** devuelve la suma de todos los elementos contenidos en la lista.	**Funciones:** `sum()`, `print()`.
47	**Verificar si todos son positivos:** Verifica si todos los elementos de `` son positivos.	`print("¿Todos positivos?:", all(n > 0 for n in numeros))`	La función **`all()`** devuelve `True` si *todos* los elementos de la expresión generadora cumplen la	**Funciones:** `all()`, `print()`.

			condición (en este caso, `n > 0`).	
48	**Verificar si hay algún número negativo:** Verifica si en `[2, -5, 7, 10]` hay al menos un número negativo.	`print("¿Hay algún negativo?:", any(n < 0 for n in numeros))`	La función **`any()`** devuelve `True` si *al menos uno* de los elementos de la expresión generadora cumple la condición (en este caso, `n < 0`).	**Funciones:** `any()`, `print()`.
49	**Eliminar elementos menores que 5:** Deja solo los números mayores o iguales a 5 en ``.	`filtrados = [n for n in numeros if n >= 5]` `print("Filtrados:", filtrados)`	Se utiliza una comprensión de lista para filtrar los elementos. Solo se incluyen aquellos que cumplen la condición **`if n >= 5`**.	**Operadores:** `>=`.
50	**Aplanar una lista de listas:** Convierte `[,,]` en ``.	`aplanada = [num for sublista in listas for num in sublista]` `print("Lista aplanada:", aplanada)`	Se utiliza una **comprensión con dos `for` anidados**. El primer `for` recorre cada `sublista`, y el segundo `for` recorre cada `num` dentro de la sublista, resultando en una lista plana.	**Estructuras:** `for` anidado dentro de lista.

Ejercicio 1: Crear una lista y mostrarla

Enunciado:
Crea una lista con los números del 1 al 5 y muéstrala.

Código:

```
numeros = [1, 2, 3, 4, 5]
print("Lista de números:", numeros)
```

Explicación:

- Se crea una lista usando corchetes `[]`.
- `print()` muestra la lista completa.

Comandos:

- Variables: `numeros`.
- Operadores: N/A.
- Funciones: `print()`.
- Estructuras: N/A.

Ejercicio 2: Acceder a elementos de la lista

Enunciado:
Muestra el primer y último elemento de la lista `[10, 20, 30, 40, 50]`.

Código:

```
numeros = [10, 20, 30, 40, 50]
print("Primer elemento:", numeros[0])
print("Último elemento:", numeros[-1])
```

Explicación:

- `numeros[0]` accede al primer elemento.
- `numeros[-1]` accede al último elemento.

Comandos:

- Variables: `numeros`.
- Operadores: Índices `[ ]`.
- Funciones: `print()`.

Ejercicio 3: Modificar elementos

Enunciado:
Cambia el tercer elemento de `[1, 2, 3, 4, 5]` a 10 y muestra la lista.

Código:

```
numeros = [1, 2, 3, 4, 5]
numeros[2] = 10
print("Lista modificada:", numeros)
```

Explicación:

- `numeros[2] = 10` reemplaza el valor del índice 2 (tercer elemento).

Comandos:

- Variables: `numeros`.
- Operadores: Índices `[ ]`, `=`.
- Funciones: `print()`.

Ejercicio 4: Agregar elementos

Enunciado:
Agrega el número 6 al final de la lista `[1, 2, 3, 4, 5]`.

Código:

```
numeros = [1, 2, 3, 4, 5]
numeros.append(6)
print("Lista después de append:", numeros)
```

Explicación:

- `append()` añade un elemento al final de la lista.

Comandos:

- Variables: `numeros`.
- Operadores: N/A.
- Funciones: `append()`, `print()`.

Ejercicio 5: Insertar elementos

Enunciado:
Inserta el número 0 al inicio de la lista `[1, 2, 3, 4, 5]`.

Código:

```
numeros = [1, 2, 3, 4, 5]
numeros.insert(0, 0)
print("Lista después de insert:", numeros)
```

Explicación:

- `insert(index, valor)` inserta un elemento en la posición indicada.

Comandos:

- Variables: `numeros`.
- Operadores: N/A.
- Funciones: `insert()`, `print()`.

Ejercicio 6: Eliminar elementos por valor

Enunciado:
Elimina el número 3 de la lista `[1, 2, 3, 4, 5]`.

Código:

```
numeros = [1, 2, 3, 4, 5]
numeros.remove(3)
print("Lista después de remove:", numeros)
```

Explicación:

- `remove(valor)` elimina el primer elemento que coincide con `valor`.

Comandos:

- Variables: `numeros`.
- Operadores: N/A.
- Funciones: `remove()`, `print()`.

Ejercicio 7: Eliminar elemento por índice

Enunciado:
Elimina el tercer elemento de `[1, 2, 3, 4, 5]` usando su índice.

Código:

```
numeros = [1, 2, 3, 4, 5]
del numeros[2]
print("Lista después de del:", numeros)
```

Explicación:

- `del` elimina un elemento por índice.

Comandos:

- Variables: `numeros`.
- Operadores: `del`, índices `[ ]`.
- Funciones: `print()`.

Ejercicio 8: Contar elementos

Enunciado:
Cuenta cuántas veces aparece el número 2 en `[1, 2, 2, 3, 2, 4]`.

Código:

```
numeros = [1, 2, 2, 3, 2, 4]
cantidad = numeros.count(2)
print("Número de veces que aparece 2:", cantidad)
```

Explicación:

- `count(valor)` devuelve cuántas veces aparece `valor` en la lista.

Comandos:

- Variables: `numeros`, `cantidad`.
- Operadores: N/A.
- Funciones: `count()`, `print()`.

Ejercicio 9: Ordenar lista

Enunciado:
Ordena de menor a mayor la lista `[5, 2, 9, 1, 7]`.

Código:

```
numeros = [5, 2, 9, 1, 7]
numeros.sort()
print("Lista ordenada:", numeros)
```

Explicación:

- `sort()` ordena los elementos de la lista en forma ascendente.

Comandos:

- Variables: `numeros`.
- Operadores: N/A.
- Funciones: `sort()`, `print()`.

Ejercicio 10: Lista inversa

Enunciado:
Invierte la lista `[1, 2, 3, 4, 5]`.

Código:

```
numeros = [1, 2, 3, 4, 5]
numeros.reverse()
print("Lista invertida:", numeros)
```

Explicación:

- `reverse()` invierte el orden de los elementos de la lista.

Comandos:

- Variables: `numeros`.
- Operadores: N/A.
- Funciones: `reverse()`, `print()`.

Ejercicios 11 a 20: Listas intermedias

Ejercicio 11: Suma de elementos de la lista

Enunciado:
Calcula la suma de todos los elementos de la lista `[1, 2, 3, 4, 5]`.

Código:

```
numeros = [1, 2, 3, 4, 5]
suma = 0
for num in numeros:
    suma += num
print("Suma total:", suma)
```

Explicación:

- Bucle `for` recorre cada elemento y lo suma a `suma`.

Comandos:

- Variables: `numeros`, `num`, `suma`.
- Operadores: `+=`.
- Funciones: `print()`.
- Estructuras: `for`.

Ejercicio 12: Promedio de elementos

Enunciado:
Calcula el promedio de los elementos de `[2, 4, 6, 8, 10]`.

Código:

```
numeros = [2, 4, 6, 8, 10]
suma = 0
for num in numeros:
    suma += num
promedio = suma / len(numeros)
print("Promedio:", promedio)
```

Explicación:

- `len(numeros)` devuelve la cantidad de elementos.
- Promedio = suma / cantidad de elementos.

Comandos:

- Variables: `numeros`, `num`, `suma`, `promedio`.
- Operadores: `+=`, `/`.
- Funciones: `len()`, `print()`.
- Estructuras: `for`.

Ejercicio 13: Encontrar el máximo

Enunciado:
Encuentra el mayor número en `[5, 7, 2, 9, 1]`.

Código:

```
numeros = [5, 7, 2, 9, 1]
mayor = numeros[0]
for num in numeros:
    if num > mayor:
        mayor = num
print("Número mayor:", mayor)
```

Explicación:

- Compara cada elemento con `mayor` y actualiza si es mayor.

Comandos:

- Variables: `numeros`, `num`, `mayor`.
- Operadores: `>`.
- Funciones: `print()`.
- Estructuras: `for`, `if`.

Ejercicio 14: Encontrar el mínimo

Enunciado:
Encuentra el menor número en `[5, 7, 2, 9, 1]`.

Código:

```
numeros = [5, 7, 2, 9, 1]
menor = numeros[0]
for num in numeros:
    if num < menor:
        menor = num
print("Número menor:", menor)
```

Explicación:

- Similar al ejercicio anterior, pero con `<`.

Comandos:

- Variables: `numeros`, `num`, `menor`.
- Operadores: `<`.
- Funciones: `print()`.
- Estructuras: `for`, `if`.

Ejercicio 15: Contar elementos mayores que un valor

Enunciado:
Cuenta cuántos números en `[2, 5, 8, 1, 9]` son mayores que 5.

Código:

```
numeros = [2, 5, 8, 1, 9]
contador = 0
for num in numeros:
    if num > 5:
        contador += 1
print("Cantidad de números mayores a 5:", contador)
```

Explicación:

- `contador` aumenta solo si el número cumple la condición.

Comandos:

- Variables: `numeros`, `num`, `contador`.
- Operadores: `>`, `+=`.
- Funciones: `print()`.
- Estructuras: `for`, `if`.

Ejercicio 16: Lista de números pares

Enunciado:
Crea una nueva lista solo con los números pares de `[1, 2, 3, 4, 5, 6]`.

Código:

```
numeros = [1, 2, 3, 4, 5, 6]
pares = []
for num in numeros:
    if num % 2 == 0:
        pares.append(num)
print("Números pares:", pares)
```

Explicación:

- `% 2 == 0` identifica números pares.
- `append()` agrega a la nueva lista.

Comandos:

- Variables: `numeros`, `num`, `pares`.
- Operadores: `%`, `==`.
- Funciones: `append()`, `print()`.
- Estructuras: `for`, `if`.

Ejercicio 17: Sumar elementos pares e impares por separado

Enunciado:
Suma los números pares e impares de `[1, 2, 3, 4, 5, 6]` por separado.

Código:

```
numeros = [1, 2, 3, 4, 5, 6]
suma_pares = 0
suma_impares = 0
for num in numeros:
    if num % 2 == 0:
        suma_pares += num
    else:
        suma_impares += num
print("Suma pares:", suma_pares)
print("Suma impares:", suma_impares)
```

Explicación:

- Se usan dos acumuladores y `if-else` para separar pares e impares.

Comandos:

- Variables: `numeros`, `num`, `suma_pares`, `suma_impares`.
- Operadores: `%`, `==`, `+=`.
- Funciones: `print()`.
- Estructuras: `for`, `if`, `else`.

Ejercicio 18: Buscar un elemento en la lista

Enunciado:
Verifica si el número 4 está en `[1, 2, 3, 4, 5]`.

Código:

```
numeros = [1, 2, 3, 4, 5]
if 4 in numeros:
    print("El número 4 está en la lista")
else:
    print("El número 4 no está en la lista")
```

Explicación:

- `in` verifica si un valor existe en la lista.

Comandos:

- Variables: `numeros`.
- Operadores: `in`.
- Funciones: `print()`.
- Estructuras: `if-else`.

Ejercicio 19: Contar ocurrencias de un elemento

Enunciado:
Cuenta cuántas veces aparece el número 2 en `[2, 3, 2, 5, 2, 1]`.

Código:

```
numeros = [2, 3, 2, 5, 2, 1]

cantidad = numeros.count(2)

print("El número 2 aparece:", cantidad, "veces")
```

Explicación:

- `count()` devuelve la cantidad de veces que aparece el valor en la lista.

Comandos:

- Variables: `numeros`, `cantidad`.
- Operadores: N/A.
- Funciones: `count()`, `print()`.

Ejercicio 20: Copiar una lista

Enunciado:
Copia la lista `[1, 2, 3, 4, 5]` a otra lista.

Código:

```
numeros = [1, 2, 3, 4, 5]
copia = numeros.copy()
print("Lista original:", numeros)
print("Copia de la lista:", copia)
```

Explicación:

- `copy()` crea una copia independiente de la lista.

Comandos:

- Variables: `numeros`, `copia`.
- Operadores: N/A.
- Funciones: `copy()`, `print()`.

Ejercicio 21: Invertir una lista

Enunciado:
Invierte la lista `[1, 2, 3, 4, 5]` para que quede `[5, 4, 3, 2, 1]`.

Código:

```
numeros = [1, 2, 3, 4, 5]

numeros.reverse()

print("Lista invertida:", numeros)
```

Explicación:

- `reverse()` invierte los elementos de la lista en su lugar.

Comandos:

- Variables: `numeros`.
- Funciones: `reverse()`, `print()`.

Ejercicio 22: Ordenar lista de menor a mayor

Enunciado:
Ordena la lista `[5, 2, 8, 1, 9]` en orden ascendente.

Código:

```
numeros = [5, 2, 8, 1, 9]

numeros.sort()

print("Lista ordenada:", numeros)
```

Explicación:

- `sort()` ordena los elementos de la lista de menor a mayor.

Comandos:

- Variables: `numeros`.
- Funciones: `sort()`, `print()`.

Ejercicio 23: Ordenar lista de mayor a menor

Enunciado:
Ordena la lista `[5, 2, 8, 1, 9]` en orden descendente.

Código:

```
numeros = [5, 2, 8, 1, 9]
numeros.sort(reverse=True)
print("Lista descendente:", numeros)
```

Explicación:

- `sort(reverse=True)` ordena de mayor a menor.

Comandos:

- Variables: `numeros`.
- Funciones: `sort()`, `print()`.

Ejercicio 24: Eliminar duplicados

Enunciado:
Elimina los números repetidos de `[1, 2, 2, 3, 4, 4, 5]`.

Código:

```
numeros = [1, 2, 2, 3, 4, 4, 5]
sin_duplicados = list(set(numeros))
print("Lista sin duplicados:", sin_duplicados)
```

Explicación:

- `set()` elimina elementos repetidos.
- `list()` convierte el conjunto de nuevo en lista.

Comandos:

- Variables: `numeros`, `sin_duplicados`.
- Funciones: `set()`, `list()`, `print()`.

Ejercicio 25: Concatenar dos listas

Enunciado:
Une las listas `[1, 2, 3]` y `[4, 5, 6]`.

Código:

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
lista_total = lista1 + lista2
print("Lista concatenada:", lista_total)
```

Explicación:

- El operador `+` concatena listas.

Comandos:

- Variables: `lista1`, `lista2`, `lista_total`.
- Operadores: `+`.
- Funciones: `print()`.

Ejercicio 26: Multiplicar lista

Enunciado:
Crea una lista que repita `[1, 2, 3]` tres veces.

Código:

```
numeros = [1, 2, 3]

repetida = numeros * 3

print("Lista repetida:", repetida)
```

Explicación:

- El operador `*` repite la lista tantas veces como se indique.

Comandos:

- Variables: `numeros`, `repetida`.
- Operadores: `*`.
- Funciones: `print()`.

Ejercicio 27: Sumar dos listas elemento por elemento

Enunciado:
Suma `[1, 2, 3]` con `[4, 5, 6]` para obtener `[5, 7, 9]`.

Código:

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
resultado = []
for i in range(len(lista1)):
    resultado.append(lista1[i] + lista2[i])
print("Suma elemento a elemento:", resultado)
```

Explicación:

- `range(len(lista1))` recorre los índices.
- Se suman posiciones correspondientes.

Comandos:

- Variables: `lista1`, `lista2`, `resultado`, `i`.
- Operadores: `+`.
- Funciones: `len()`, `range()`, `append()`, `print()`.
- Estructuras: `for`.

Ejercicio 28: Lista con cuadrados de números

Enunciado:
Crea una lista con los cuadrados de `[1, 2, 3, 4, 5]`.

Código:

```
numeros = [1, 2, 3, 4, 5]
cuadrados = []
for num in numeros:
    cuadrados.append(num ** 2)
print("Cuadrados:", cuadrados)
```

Explicación:

- `** 2` calcula el cuadrado de cada número.

Comandos:

- Variables: `numeros`, `cuadrados`, `num`.
- Operadores: `**`.
- Funciones: `append()`, `print()`.
- Estructuras: `for`.

Ejercicio 29: Intercambiar dos elementos

Enunciado:
Intercambia el primer y último elemento de `[1, 2, 3, 4, 5]`.

Código:

```
numeros = [1, 2, 3, 4, 5]
numeros[0], numeros[-1] = numeros[-1], numeros[0]
print("Lista modificada:", numeros)
```

Explicación:

- Intercambio múltiple: `a, b = b, a`.

Comandos:

- Variables: `numeros`.
- Operadores: asignación múltiple.
- Funciones: `print()`.

Ejercicio 30: Lista con números al revés usando slicing

Enunciado:
Invierte `[1, 2, 3, 4, 5]` usando slicing.

Código:

```
numeros = [1, 2, 3, 4, 5]
invertida = numeros[::-1]
print("Lista invertida con slicing:", invertida)
```

Explicación:

- `[::-1]` recorre la lista en orden inverso.

Comandos:

- Variables: `numeros`, `invertida`.
- Funciones: `print()`.
- Operadores: slicing (`[inicio:fin:paso]`).

Ejercicio 31: Crear una matriz 3x3

Enunciado:
Crea una matriz de 3x3 con los números del 1 al 9.

Código:

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print("Matriz 3x3:")
for fila in matriz:
    print(fila)
```

Explicación:

- Una lista dentro de otra lista forma una **matriz**.
- Bucle `for` recorre cada fila.

Comandos:

- Variables: `matriz`, `fila`.
- Estructuras: lista de listas, `for`.
- Funciones: `print()`.

Ejercicio 32: Acceder a un elemento de la matriz

Enunciado:
Muestra el número en la segunda fila, tercera columna de la matriz anterior.

Código:

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print("Elemento en fila 2, columna 3:", matriz[1][2])
```

Explicación:

- `matriz[1][2]` → fila 2 (índice 1), columna 3 (índice 2).

Comandos:

- Variables: `matriz`.
- Operadores: índices dobles `[i][j]`.
- Funciones: `print()`.

Ejercicio 33: Recorrer todos los elementos de la matriz

Enunciado:
Muestra cada número de la matriz 3x3 por separado.

Código:

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
for fila in matriz:
    for elemento in fila:
        print(elemento, end=" ")
```

Explicación:

- Bucle anidado: uno recorre filas y otro columnas.
- `end=" "` imprime en la misma línea.

Comandos:

- Variables: `matriz`, `fila`, `elemento`.
- Estructuras: `for` anidados.
- Funciones: `print()`.

Ejercicio 34: Sumar todos los elementos de la matriz

Enunciado:
Suma todos los elementos de la matriz 3x3.

Código:

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
suma = 0
for fila in matriz:
    for elemento in fila:
        suma += elemento
print("Suma total:", suma)
```

Explicación:

- Se acumulan los valores con `+=`.

Comandos:

- Variables: `suma`, `fila`, `elemento`.
- Operadores: `+=`.
- Estructuras: `for` anidado.

Ejercicio 35: Crear una lista con los números pares de la matriz

Enunciado:
Guarda en una lista todos los números pares de la matriz 3x3.

Código:

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
pares = []
for fila in matriz:
    for elemento in fila:
        if elemento % 2 == 0:
            pares.append(elemento)
print("Números pares:", pares)
```

Explicación:

- Se verifica si es par (`% 2 == 0`).
- `append()` guarda en la lista.

Comandos:

- Variables: `pares`, `elemento`.
- Operadores: `%`, `==`.
- Estructuras: `for`, `if`.

Ejercicio 36: Transponer una matriz

Enunciado:
Transforma la matriz 3x3 en su transpuesta (filas → columnas).

Código:

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
transpuesta = []
for i in range(len(matriz[0])):
    fila = []
    for j in range(len(matriz)):
        fila.append(matriz[j][i])
    transpuesta.append(fila)
print("Matriz transpuesta:")
for fila in transpuesta:
    print(fila)
```

Explicación:

- Intercambia filas por columnas.
- `matriz[j][i]` accede al valor correcto.

Comandos:

- Variables: `i`, `j`, `fila`, `transpuesta`.
- Funciones: `len()`, `append()`.
- Estructuras: `for` anidados.

Ejercicio 37: Generar lista con los cuadrados de una matriz

Enunciado:
Crea una nueva matriz con los cuadrados de cada número.

Código:

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
cuadrados = []
for fila in matriz:
    nueva_fila = []
    for elemento in fila:
        nueva_fila.append(elemento ** 2)
    cuadrados.append(nueva_fila)
print("Matriz de cuadrados:")
for fila in cuadrados:
    print(fila)
```

Explicación:

- Cada elemento se eleva al cuadrado (`** 2`).

Comandos:

- Variables: `cuadrados`, `nueva_fila`.
- Operadores: `**`.
- Estructuras: `for` anidados.

Ejercicio 38: Sumar dos matrices

Enunciado:
Suma estas dos matrices:

```
A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
B = [[9, 8, 7], [6, 5, 4], [3, 2, 1]]
```

Código:

```
A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
B = [[9, 8, 7], [6, 5, 4], [3, 2, 1]]
suma = []
for i in range(len(A)):
    fila = []
    for j in range(len(A[0])):
        fila.append(A[i][j] + B[i][j])
    suma.append(fila)
print("Suma de matrices:")
for fila in suma:
    print(fila)
```

Explicación:

- Se suman elementos en la misma posición `[i][j]`.

Comandos:

- Variables: `A`, `B`, `suma`, `i`, `j`.
- Funciones: `len()`, `append()`.
- Estructuras: `for` anidados.

Ejercicio 39: Buscar un número en la matriz

Enunciado:
Verifica si el número `5` está en la matriz.

Código:

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
encontrado = False
for fila in matriz:
    if 5 in fila:
        encontrado = True
        break
if encontrado:
    print("El número 5 está en la matriz")
else:
    print("El número 5 no está en la matriz")
```

Explicación:

- `in` verifica si `5` está en la fila.
- `break` detiene la búsqueda.

Comandos:

- Variables: `encontrado`.
- Operadores: `in`.
- Estructuras: `for`, `if`, `break`.

Ejercicio 40: Calcular la diagonal principal de una matriz

Enunciado:
Muestra los elementos de la diagonal principal de la matriz 3x3.

Código:

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
diagonal = []
for i in range(len(matriz)):
    diagonal.append(matriz[i][i])
print("Diagonal principal:", diagonal)
```

Explicación:

- La diagonal principal son los elementos `[0][0]`, `[1][1]`, `[2][2]`.

Comandos:

- Variables: `diagonal`, `i`.
- Funciones: `len()`, `append()`.
- Estructuras: `for`.

Ejercicio 41: Crear una lista de números del 1 al 10 con comprensión

Enunciado:
Genera automáticamente la lista `[1, 2, 3, ..., 10]`.

Código:

```
numeros = [i for i in range(1, 11)]
print("Lista generada:", numeros)
```

Explicación:

- Comprensión de listas: `[i for i in range(1, 11)]`.
- `range(1, 11)` genera del 1 al 10.

Comandos:

- Funciones: `range()`, `print()`.
- Operadores: `for` dentro de lista.

Ejercicio 42: Lista de cuadrados con comprensión

Enunciado:
Crea una lista con los cuadrados de `[1, 2, 3, 4, 5]`.

Código:

```
numeros = [1, 2, 3, 4, 5]
cuadrados = [n**2 for n in numeros]
print("Cuadrados:", cuadrados)
```

Explicación:

- Se calcula `n**2` dentro de la comprensión.

Comandos:

- Variables: `numeros`, `cuadrados`.
- Operadores: `**`.

Ejercicio 43: Filtrar números pares con comprensión

Enunciado:
Crea una lista con solo los números pares de `[1, 2, 3, 4, 5, 6]`.

Código:

```
numeros = [1, 2, 3, 4, 5, 6]
pares = [n for n in numeros if n % 2 == 0]
print("Números pares:", pares)
```

Explicación:

- `if n % 2 == 0` filtra pares.

Comandos:

- Operadores: `%`, `==`.
- Estructuras: `if` en comprensión.

Ejercicio 44: Elevar a la tercera potencia solo los impares

Enunciado:
Crea una lista con el cubo de los números impares en `[1, 2, 3, 4, 5]`.

Código:

```
numeros = [1, 2, 3, 4, 5]
cubos_impares = [n**3 for n in numeros if n % 2 != 0]
print("Cubos de impares:", cubos_impares)
```

Explicación:

- `n**3` calcula el cubo.
- `if n % 2 != 0` selecciona impares.

Comandos:

- Operadores: `%`, `!=`, `**`.

Ejercicio 45: Usar `max` y `min` en listas

Enunciado:
Encuentra el mayor y menor número de `[5, 9, 2, 8, 1]`.

Código:

```
numeros = [5, 9, 2, 8, 1]
print("Máximo:", max(numeros))
print("Mínimo:", min(numeros))
```

Explicación:

- `max(lista)` devuelve el mayor.
- `min(lista)` devuelve el menor.

Comandos:

- Funciones: `max()`, `min()`, `print()`.

Ejercicio 46: Usar `sum` en listas

Enunciado:
Calcula la suma total de `[3, 6, 9, 12]`.

Código:

```
numeros = [3, 6, 9, 12]

print("Suma total:", sum(numeros))
```

Explicación:

- `sum(lista)` devuelve la suma de todos los elementos.

Comandos:

- Funciones: `sum()`, `print()`.

Ejercicio 47: Verificar si todos son positivos

Enunciado:
Verifica si todos los elementos de `[2, 4, 6, 8]` son positivos.

Código:

```
numeros = [2, 4, 6, 8]

print("¿Todos positivos?:", all(n > 0 for n in numeros))
```

Explicación:

- `all()` devuelve `True` si todos cumplen la condición.

Comandos:

- Funciones: `all()`, `print()`.
- Expresiones: `n > 0 for n in numeros`.

Ejercicio 48: Verificar si hay algún número negativo

Enunciado:
Verifica si en `[2, -5, 7, 10]` hay al menos un número negativo.

Código:

```
numeros = [2, -5, 7, 10]
print("¿Hay algún negativo?:", any(n < 0 for n in numeros))
```

Explicación:

- `any()` devuelve `True` si al menos uno cumple la condición.

Comandos:

- Funciones: `any()`, `print()`.
- Expresiones: `n < 0 for n in numeros`.

Ejercicio 49: Eliminar elementos menores que 5

Enunciado:
Deja solo los números mayores o iguales a 5 en `[3, 5, 7, 2, 8]`.

Código:

```
numeros = [3, 5, 7, 2, 8]
filtrados = [n for n in numeros if n >= 5]
print("Filtrados:", filtrados)
```

Explicación:

- Se aplica condición `if n >= 5` dentro de la comprensión.

Comandos:

- Operadores: `>=`.

Ejercicio 50: Aplanar una lista de listas

Enunciado:
Convierte `[[1, 2], [3, 4], [5, 6]]` en `[1, 2, 3, 4, 5, 6]`.

Código:

```
listas = [[1, 2], [3, 4], [5, 6]]
aplanada = [num for sublista in listas for num in sublista]
print("Lista aplanada:", aplanada)
```

Explicación:

- Comprensión con dos `for`: primero recorre cada sublista, luego cada número.

Comandos:

- Variables: `listas`, `aplanada`, `sublista`, `num`.
- Estructuras: `for` anidado dentro de lista.

MANUAL DETALLADO DE EJERCICIOS DE ESTRUCTURAS DE CONJUNTOS

Ejercicios Fundamentales y Manipulación Básica

Ejercicio	Enunciado y Código	Explicación Minuciosa	Comandos y Controles
1	**Crea un conjunto con los números del 1 al 5 e imprímelo**.	Creamos el conjunto utilizando las llaves `{}` para definir sus elementos y luego lo mostramos en la salida usando la función `print()`.	`{}` → define un conjunto. `print()` → muestra el conjunto.
2	**Crea un conjunto vacío y añade los números 1, 2 y 3**.	El conjunto vacío se crea con la función `set()`. Para ir añadiendo elementos individualmente se utiliza el método `.add()`.	`set()` → conjunto vacío. `.add(x)` → agrega un elemento.
3	**Comprueba si el número 2 está en el conjunto {1,2,3,4,5}**.	Se usa el operador `in` para verificar si un elemento específico es miembro del conjunto. Si lo es, el operador devuelve el valor booleano `True`.	`in` → operador de pertenencia.
4	**Elimina el número 3 de {1,2,3,4,5}**.	El método `.remove()` se utiliza para borrar un elemento indicado del conjunto. Es crucial notar que si el elemento que se intenta eliminar no existe en el conjunto, este método arrojará un error.	`.remove(x)` → elimina un elemento (produce error si no existe).
5	**Usa .discard() para eliminar el número 10 de {1,2,3,4,5}**.	A diferencia de `.remove()`, el método `.discard()` se utiliza para eliminar un elemento, pero **no genera un error** (`KeyError`) si el elemento especificado no se encuentra en el conjunto.	`.discard(x)` → elimina el elemento si existe.
6	**Usa .pop() para extraer un elemento aleatorio de {10,20,30,40}**.	El método `.pop()` cumple una doble función: elimina un elemento del conjunto y, simultáneamente, devuelve ese elemento extraído de manera **aleatoria**.	`.pop()` → saca un elemento al azar.
7	**Borra todos los elementos de {1,2,3,4,5} con .clear()**.	Después de ejecutar el método `.clear()`, todos los elementos son retirados, dejando el conjunto completamente vacío.	`.clear()` → vacía el conjunto.
24	**Copia un conjunto {1,2,3} en otro conjunto**.	Para asegurar que la copia sea independiente (copia superficial), se utiliza el método `.copy()`. Si se modifica la copia, el original no se ve afectado.	`.copy()` → duplica el conjunto.
26	**Añade múltiples elementos {4,5,6} a un conjunto {1,2,3}**.	El método `.update()` permite agregar varios elementos de una vez, ya sea provenientes de otro conjunto o de cualquier objeto iterable (como una lista).	`.update(iterable)` → agrega varios elementos.

Ejercicios de Operaciones de Conjuntos (Métodos)

Ejercicio	Enunciado y Código	Explicación Minuciosa	Comandos y Controles
8	**Crea dos conjuntos {1,2,3} y {3,4,5} y calcula su unión**.	La unión combina todos los elementos presentes en ambos conjuntos, eliminando automáticamente cualquier elemento repetido. Se usa el método `.union()`.	`.union(otro)` → devuelve la unión.
9	**Encuentra la intersección de {1,2,3} y {2,3,4}**.	La intersección tiene como resultado un nuevo conjunto que contiene únicamente los elementos que son comunes (compartidos) entre los conjuntos de entrada.	`.intersection(otro)` → devuelve comunes.
10	**Encuentra la diferencia entre {1,2,3,4} y {3,4,5}**.	La diferencia devuelve solamente los elementos que pertenecen al primer conjunto (`a`) pero que no se encuentran en el segundo conjunto (`b`).	`.difference(otro)` → devuelve la diferencia.
11	**Encuentra la diferencia simétrica entre {1,2,3} y {3,4,5}**.	La diferencia simétrica obtiene todos los elementos que están en un conjunto o en el otro, pero **excluyendo** aquellos que son comunes a ambos (la intersección).	`.symmetric_difference(otro)` → diferencia simétrica.

Ejercicios de Operaciones de Conjuntos (Operadores)

Ejercicio	Enunciado y Código	Explicación Minuciosa	Comandos y Controles
12	**Usa el operador \	para unir {1,2} y {2,3}**.	El operador pipe, `|`,
13	**Usa el operador & para calcular la intersección de {1,2,3} y {2,3,4}**.	El operador ampersand, `&`, es la forma abreviada para encontrar los elementos comunes, siendo equivalente al método `.intersection()`.	`&` → intersección de conjuntos.
14	**Usa el operador - para calcular la diferencia de {1,2,3,4} menos {3,4}**.	El operador de resta, `-`, elimina del primer conjunto (`a`) todos los elementos que están presentes en el segundo conjunto (`b`).	`-` → diferencia de conjuntos.
15	**Usa el operador ^ para calcular la diferencia simétrica de {1,2,3} y {3,4,5}**.	El operador caret, `^`, funciona como `.symmetric_difference()`, devolviendo los elementos que son únicos en cada uno de los conjuntos, sin incluir los comunes.	`^` → diferencia simétrica.
43	**Elimina los elementos de {1,2,3,4,5} que**	El operador de asignación de diferencia, `-=`, modifica el conjunto original (`a`) eliminando directamente los elementos comunes con el otro	`-=` → diferencia en asignación (modifica el

	también estén en {2,4} usando diferencia.	conjunto (b).	conjunto).

Ejercicios de Relaciones y Pertenencia

Ejercicio	Enunciado y Código	Explicación Minuciosa	Comandos y Controles
16	**Comprueba si {1,2} es subconjunto de {1,2,3,4}**.	Un subconjunto contiene solo elementos que ya están incluidos en el conjunto más grande. Se verifica con el método `.issubset()`.	`.issubset(otro)` → verifica si es subconjunto.
17	**Comprueba si {1,2,3,4} es superconjunto de {2,3}**.	Un superconjunto es aquel que incluye a la totalidad de los elementos del otro conjunto. Se verifica con el método `.issuperset()`.	`.issuperset(otro)` → verifica si contiene al otro conjunto.
18	**Comprueba si {1,2} y {3,4} son conjuntos disjuntos**.	Los conjuntos se consideran disjuntos si no tienen absolutamente ningún elemento en común, es decir, si su intersección es un conjunto vacío. Se verifica con `.isdisjoint()`.	`.isdisjoint(otro)` → `True` si no hay intersección.
41	**Verifica si todos los elementos de {1,2,3} están dentro de {0,1,2,3,4,5} usando <=.**	El operador de comparación `<=` (menor o igual que) se utiliza como una alternativa para comprobar si un conjunto es subconjunto de otro (igual que `.issubset()`).	`<=` → operador de subconjunto.
42	**Verifica si {1,2,3,4,5} contiene a {2,3} usando >=.**	El operador de comparación `>=` (mayor o igual que) se usa para verificar si un conjunto es superconjunto del otro (igual que `.issuperset()`).	`>=` → operador de superconjunto.

Ejercicios de Conversión, Iteración y Utilidad

Ejercicio	Enunciado y Código	Explicación Minuciosa	Comandos y Controles
19	**Crea un conjunto a partir de una lista y observa el resultado**.	Al convertir una lista en un conjunto usando `set(lista)`, la característica intrínseca de los conjuntos de Python es que **eliminan automáticamente los elementos duplicados**.	`set(lista)` → convierte lista en conjunto, quitando repetidos.
20	**Convierte un conjunto {1,2,3,4} en una lista**.	La conversión a lista, usando `list(conjunto)`, es útil si se requiere ordenar los elementos o si se necesita acceder a ellos mediante índices posicionales.	`list(conjunto)` → convierte conjunto en lista.
21	**Ordena un conjunto {5,3,1,4,2} convirtiéndolo en**	Los conjuntos no mantienen un orden interno. La función `sorted(conjunto)` devuelve una lista que contiene los elementos del conjunto en orden	`sorted(conjunto)` → devuelve lista ordenada.

	lista.	ascendente.	
22	**Itera sobre un conjunto {10,20,30} e imprime sus elementos**.	Es posible recorrer todos los elementos de un conjunto utilizando una estructura de bucle `for`.	`for elemento in conjunto:` → recorre todos los elementos.
23	**Calcula cuántos elementos hay en el conjunto {1,2,3,4,5}**.	La función incorporada `len()` se utiliza para obtener el número total de elementos que contiene el conjunto.	`len(conjunto)` → cuenta elementos.

Ejercicios de Manejo de Errores y Comprensión

Ejercicio	Enunciado y Código	Explicación Minuciosa	Comandos y Controles
25	**Elimina un elemento con remove() y atrapa el error si no existe**.	Cuando se intenta usar `.remove()` en un elemento no existente, se lanza un error `KeyError`. Se utiliza la estructura `try-except` para capturar y manejar este error sin detener el programa.	`try-except` → manejo de errores.
27	**Usa comprensión de conjuntos para crear un conjunto con los cuadrados de 1 al 5**.	La **comprensión de conjuntos** (`set comprehension`) permite generar automáticamente conjuntos aplicando una expresión (`x**2`) a cada elemento de un iterable (`range(1, 6)`).	`{expresion for var in iterable}` → comprensión de conjuntos. `x**2` → potencia al cuadrado.
31	**Crea un conjunto con los números pares del 1 al 10 usando comprensión de conjuntos**.	Se utiliza la comprensión de conjuntos con una cláusula de filtro (`if x % 2 == 0`) para incluir solo los números del rango que son divisibles por 2 (pares).	`{expresion for var in iterable if condicion}` → comprensión de conjuntos con filtro. `%` → operador módulo (resto).
34	**Crea un conjunto con los múltiplos de 3 menores que 20**.	Se utiliza `range(20)` para iterar los números del 0 al 19 y se aplica el operador módulo `%` para filtrar aquellos cuyo resto al dividirse por 3 es cero.	`range(20)` → números del 0 al 19. `%` → módulo.
35	**Dado el conjunto {1,2,3,4,5}, crea otro con sus elementos elevados al cubo**.	Se usa la comprensión de conjuntos, aplicando la expresión de potencia al cubo (`x**3`) a cada elemento del conjunto original.	`x**3` → potencia al cubo.
37	**Crea un conjunto con los números impares de 1 a 15**.	La comprensión de conjuntos filtra aquellos números en el rango que, al dividirse por 2, tienen un resto diferente de cero (`x % 2 != 0`).	`% 2 != 0` → condición de imparidad.
40	**Dado el conjunto {1,2,3,4,5}, elimina los números pares**.	Se reasigna el conjunto utilizando una comprensión que filtra los elementos, manteniendo solo aquellos que cumplen la condición de	`if x % 2 != 0` → condición para filtrar impares.

		ser impares (`if x % 2 != 0`).	
49	**Crea un conjunto con los divisores de 12**.	Se emplea la comprensión de conjuntos para iterar desde 1 hasta 12 y se verifica qué números cumplen la condición de ser un divisor exacto de 12 (`if 12 % x == 0`).	`if 12 % x == 0` → condición de divisor.

Ejercicios de Aplicación Práctica y Casos Específicos

Ejercicio	Enunciado y Código	Explicación Minuciosa	Comandos y Controles
28	**Encuentra los caracteres únicos en la palabra "banana"**.	Al convertir una cadena (`string`) directamente a un conjunto usando `set(cadena)`, se obtiene automáticamente un conjunto con todas las letras (caracteres) sin repetir.	`set(cadena)` → convierte string en conjunto de caracteres.
29	**Comprueba si dos listas tienen algún elemento en común usando conjuntos**.	Se convierten ambas listas a conjuntos y se utiliza el operador de intersección (&). Si el resultado es un conjunto vacío, no hay elementos comunes.	`set(lista)` → convierte lista a conjunto. & → intersección.
30	**Usa conjuntos para eliminar duplicados de la lista**.	Se aprovecha la característica de eliminación de duplicados del conjunto y luego se reconvierte el resultado a una lista para mantener el formato original.	`set(lista)` → elimina repetidos. `list(conjunto)` → convierte a lista.
32	**Encuentra la unión de tres conjuntos {1,2}, {2,3}, {3,4}**.	El operador de unión `	` puede encadenarse para calcular la unión de más de dos conjuntos de forma simultánea.
33	**Encuentra los elementos comunes en tres conjuntos {1,2,3}, {2,3,4}, {3,4,5}**.	Similar a la unión, el operador de intersección & puede encadenarse para hallar los elementos que son comunes a la totalidad de los conjuntos implicados.	& → operador de intersección.
36	**Une dos listas eliminando duplicados con conjuntos. Listas: y**.	Se convierten ambas listas a conjuntos para eliminar duplicados, se realiza la unión con `	`, y finalmente se convierte el resultado de nuevo a lista.
38	**Comprueba si dos listas contienen exactamente los mismos elementos ignorando duplicados. Listas: y**.	Al convertir ambas listas a conjuntos, el orden y los duplicados dejan de ser relevantes. La comparación de igualdad (==) verifica si ambos conjuntos resultantes son idénticos.	== → compara igualdad.
39	**Calcula la cantidad de elementos únicos de la lista**.	Se usa `set()` para reducir la lista a elementos únicos y luego `len()` para contarlos.	`set(lista)` → elimina repetidos. `len()` → cuenta elementos.

44	**Encuentra los elementos únicos entre dos listas usando diferencia simétrica. Listas: y**.	Se convierten las listas a conjuntos y el operador de diferencia simétrica ^ devuelve los elementos que son exclusivos de cada una de las listas originales.	`^` → diferencia simétrica.
45	**Convierte un texto en conjunto de palabras únicas. Texto: "python es divertido y python es útil"**.	Primero, el método `.split()` divide el texto en una lista de palabras. Luego, `set()` elimina las palabras repetidas, dejando solo las únicas.	`.split()` → separa palabras. `set(lista)` → elimina duplicados.
46	**Usa frozenset para crear un conjunto inmutable con {1,2,3}**.	Un `frozenset` es conceptualmente idéntico a un conjunto estándar (`set`), pero su principal característica es que **no puede ser modificado** después de su creación (es inmutable).	`frozenset()` → conjunto inmutable.
47	**Encuentra los caracteres únicos de dos cadenas y únelos. Cadenas: "hola" y "mundo"**.	Se convierten ambas cadenas en conjuntos de caracteres individuales y luego se utiliza el operador de unión `	` para combinarlos.
48	**Encuentra las letras comunes entre "python" y "java"**.	Tras convertir ambas palabras a conjuntos de letras, se aplica la intersección (`&`) para identificar aquellos caracteres que aparecen en ambas cadenas.	`&` → intersección.
50	**Usa conjuntos para verificar si una palabra es un isograma (sin letras repetidas). Palabra: "murcielago"**.	Una palabra es un isograma si la cantidad de letras únicas (obtenidas con `set()`) es exactamente igual a la longitud total de la palabra.	`set(palabra)` → letras únicas. `len()` → cuenta elementos. `==` → comparación de igualdad.

Ejercicio 1

Enunciado: Crea un conjunto con los números del 1 al 5 e imprímelo.

```
numeros = {1, 2, 3, 4, 5}
print(numeros)
```

Explicación:
Creamos un conjunto con `{}` y lo mostramos.

Comandos:

- `{}` → define un conjunto.
- `print()` → muestra el conjunto.

Ejercicio 2

Enunciado: Crea un conjunto vacío y añade los números 1, 2 y 3.

```
conjunto = set()
conjunto.add(1)
conjunto.add(2)
conjunto.add(3)
print(conjunto)
```

Explicación:
Creamos un conjunto vacío con `set()` y vamos añadiendo elementos con `.add()`.

Comandos:

- `set()` → conjunto vacío.
- `.add(x)` → agrega un elemento.

Ejercicio 3

Enunciado: Comprueba si el número 2 está en el conjunto `{1,2,3,4,5}`.

```
numeros = {1, 2, 3, 4, 5}

print(2 in numeros)
```

Explicación:
El operador `in` devuelve `True` si el elemento está dentro.

Comandos:

- `in` → operador de pertenencia.

Ejercicio 4

Enunciado: Elimina el número 3 de `{1,2,3,4,5}`.

```
numeros = {1, 2, 3, 4, 5}

numeros.remove(3)

print(numeros)
```

Explicación:
`.remove()` borra el elemento indicado.

Comandos:

- `.remove(x)` → elimina un elemento (error si no existe).

Ejercicio 5

Enunciado: Usa `.discard()` para eliminar el número 10 de `{1,2,3,4,5}`.

```
numeros = {1, 2, 3, 4, 5}

numeros.discard(10)

print(numeros)
```

Explicación:
`.discard()` no da error si el elemento no está.

Comandos:

- `.discard(x)` → elimina el elemento si existe.

Ejercicio 6

Enunciado: Usa `.pop()` para extraer un elemento aleatorio de `{10,20,30,40}`.

```
conjunto = {10, 20, 30, 40}
extraido = conjunto.pop()
print("Elemento eliminado:", extraido)
print("Conjunto actual:", conjunto)
```

Explicación:
`.pop()` elimina y devuelve un elemento al azar.

Comandos:

- `.pop()` → saca un elemento al azar.

Ejercicio 7

Enunciado: Borra todos los elementos de `{1,2,3,4,5}` con `.clear()`.

```
numeros = {1, 2, 3, 4, 5}

numeros.clear()

print(numeros)
```

Explicación:
El conjunto queda vacío tras usar `.clear()`.

Comandos:

- `.clear()` → vacía el conjunto.

Ejercicio 8

Enunciado: Crea dos conjuntos `{1,2,3}` y `{3,4,5}` y calcula su unión.

```
a = {1, 2, 3}
b = {3, 4, 5}
union = a.union(b)
print(union)
```

Explicación:
La unión combina elementos de ambos conjuntos sin repetidos.

Comandos:

- `.union(otro)` → devuelve la unión.

Ejercicio 9

Enunciado: Encuentra la intersección de `{1,2,3}` y `{2,3,4}`.

```
a = {1, 2, 3}
b = {2, 3, 4}
interseccion = a.intersection(b)
print(interseccion)
```

Explicación:
La intersección obtiene los elementos comunes.

Comandos:

- `.intersection(otro)` → devuelve comunes.

Ejercicio 10

Enunciado: Encuentra la diferencia entre `{1,2,3,4}` y `{3,4,5}`.

```
a = {1, 2, 3, 4}
b = {3, 4, 5}
diferencia = a.difference(b)
print(diferencia)
```

Explicación:
La diferencia devuelve los elementos de `a` que no están en `b`.

Comandos:

- `.difference(otro)` → devuelve la diferencia.

Ejercicio 11

Enunciado: Encuentra la diferencia simétrica entre `{1,2,3}` y `{3,4,5}`.

```
a = {1, 2, 3}
b = {3, 4, 5}
simetrica = a.symmetric_difference(b)
print(simetrica)
```

Explicación:
La diferencia simétrica devuelve los elementos que están en un conjunto u otro, pero no en ambos.

Comandos:

- `.symmetric_difference(otro)` → diferencia simétrica.

Ejercicio 12

Enunciado: Usa el operador `|` para unir `{1,2}` y `{2,3}`.

```
a = {1, 2}
b = {2, 3}
union = a | b
print(union)
```

Explicación:
El operador `|` hace lo mismo que `.union()`.

Comandos:

- `|` → unión de conjuntos.

Ejercicio 13

Enunciado: Usa el operador & para calcular la intersección de `{1,2,3}` y `{2,3,4}`.

```
a = {1, 2, 3}
b = {2, 3, 4}
interseccion = a & b
print(interseccion)
```

Explicación:
El operador & es equivalente a `.intersection()`.

Comandos:

- & → intersección de conjuntos.

Ejercicio 14

Enunciado: Usa el operador `-` para calcular la diferencia de `{1,2,3,4}` menos `{3,4}`.

```
a = {1, 2, 3, 4}
b = {3, 4}
diferencia = a - b
print(diferencia)
```

Explicación:
El operador `-` elimina los elementos de `b` en `a`.

Comandos:

- `-` → diferencia de conjuntos.

Ejercicio 15

Enunciado: Usa el operador ^ para calcular la diferencia simétrica de `{1,2,3}` y `{3,4,5}`.

```
a = {1, 2, 3}
b = {3, 4, 5}
simetrica = a ^ b
print(simetrica)
```

Explicación:
El operador ^ equivale a `.symmetric_difference()`.

Comandos:

- ^ → diferencia simétrica.

Ejercicio 16

Enunciado: Comprueba si `{1,2}` es subconjunto de `{1,2,3,4}`.

```
a = {1, 2}
b = {1, 2, 3, 4}
print(a.issubset(b))
```

Explicación:
Un subconjunto contiene solo elementos que están en el conjunto mayor.

Comandos:

- `.issubset(otro)` → verifica si es subconjunto.

Ejercicio 17

Enunciado: Comprueba si `{1,2,3,4}` es superconjunto de `{2,3}`.

```
a = {1, 2, 3, 4}
b = {2, 3}
print(a.issuperset(b))
```

Explicación:
Un superconjunto incluye a todos los elementos del otro.

Comandos:

- `.issuperset(otro)` → verifica si contiene al otro conjunto.

Ejercicio 18

Enunciado: Comprueba si `{1,2}` y `{3,4}` son conjuntos disjuntos.

```
a = {1, 2}
b = {3, 4}
print(a.isdisjoint(b))
```

Explicación:
Los conjuntos disjuntos no tienen elementos en común.

Comandos:

- `.isdisjoint(otro)` → `True` si no hay intersección.

Ejercicio 19

Enunciado: Crea un conjunto a partir de una lista `[1,2,2,3,4,4,5]` y observa el resultado.

```
lista = [1, 2, 2, 3, 4, 4, 5]
conjunto = set(lista)
print(conjunto)
```

Explicación:
Un conjunto elimina duplicados automáticamente.

Comandos:

- `set(lista)` → convierte lista en conjunto, quitando repetidos.

Ejercicio 20

Enunciado: Convierte un conjunto `{1,2,3,4}` en una lista.

```
conjunto = {1, 2, 3, 4}

lista = list(conjunto)

print(lista)
```

Explicación:
Podemos convertir un conjunto a lista para poder ordenar o acceder por índices.

Comandos:

- `list(conjunto)` → convierte conjunto en lista.

Ejercicio 21

Enunciado: Ordena un conjunto `{5,3,1,4,2}` convirtiéndolo en lista.

```
conjunto = {5, 3, 1, 4, 2}

lista_ordenada = sorted(conjunto)

print(lista_ordenada)
```

Explicación:
Los conjuntos no están ordenados, pero con `sorted()` podemos obtener una lista ordenada.

Comandos:

- `sorted(conjunto)` → devuelve lista ordenada.

Ejercicio 22

Enunciado: Itera sobre un conjunto `{10,20,30}` e imprime sus elementos.

```
conjunto = {10, 20, 30}
for numero in conjunto:
    print(numero)
```

Explicación:
Podemos recorrer un conjunto con un `for`.

Comandos:

- `for elemento in conjunto:` → recorre todos los elementos.

Ejercicio 23

Enunciado: Calcula cuántos elementos hay en el conjunto `{1,2,3,4,5}`.

```
conjunto = {1, 2, 3, 4, 5}
print(len(conjunto))
```

Explicación:
`len()` devuelve el número de elementos.

Comandos:

- `len(conjunto)` → cuenta elementos.

Ejercicio 24

Enunciado: Copia un conjunto `{1,2,3}` en otro conjunto.

```
a = {1, 2, 3}
b = a.copy()
print(b)
```

Explicación:
`.copy()` crea una copia independiente.

Comandos:

- `.copy()` → duplica el conjunto.

Ejercicio 25

Enunciado: Elimina un elemento con `remove()` y atrapa el error si no existe.

```
conjunto = {1, 2, 3}
try:
    conjunto.remove(5)
except KeyError:
    print("El elemento no existe")
```

Explicación:
Si el elemento no está, `remove()` lanza un error `KeyError`, que atrapamos con `try-except`.

Comandos:

- `try-except` → manejo de errores.

Ejercicio 26

Enunciado: Añade múltiples elementos `{4,5,6}` a un conjunto `{1,2,3}`.

```
conjunto = {1, 2, 3}

conjunto.update({4, 5, 6})

print(conjunto)
```

Explicación:
`.update()` añade varios elementos de otro conjunto o lista.

Comandos:

- `.update(iterable)` → agrega varios elementos.

Ejercicio 27

Enunciado: Usa comprensión de conjuntos para crear un conjunto con los cuadrados de 1 al 5.

```
cuadrados = {x**2 for x in range(1, 6)}
print(cuadrados)
```

Explicación:
Usamos una *set comprehension* para generar automáticamente los cuadrados.

Comandos:

- `{expresion for var in iterable}` → comprensión de conjuntos.
- `x**2` → potencia al cuadrado.

Ejercicio 28

Enunciado: Encuentra los caracteres únicos en la palabra `"banana"`.

```
palabra = "banana"

unicos = set(palabra)

print(unicos)
```

Explicación:
Al convertir una cadena en conjunto, se eliminan las letras repetidas.

Comandos:

- `set(cadena)` → convierte string en conjunto de caracteres.

Ejercicio 29

Enunciado: Comprueba si dos listas tienen algún elemento en común usando conjuntos.

```
lista1 = [1, 2, 3]
lista2 = [3, 4, 5]
comunes = set(lista1) & set(lista2)
print(comunes)
```

Explicación:
Convertimos ambas listas a conjuntos y usamos & para la intersección.

Comandos:

- `set(lista)` → convierte lista a conjunto.
- `&` → intersección.

Ejercicio 30

Enunciado: Usa conjuntos para eliminar duplicados de la lista `[1,1,2,2,3,3]`.

```
lista = [1, 1, 2, 2, 3, 3]
sin_duplicados = list(set(lista))
print(sin_duplicados)
```

Explicación:
Un conjunto elimina duplicados, luego convertimos de nuevo a lista.

Comandos:

- `set(lista)` → elimina repetidos.
- `list(conjunto)` → convierte a lista.

Ejercicio 31

Enunciado: Crea un conjunto con los números pares del 1 al 10 usando comprensión de conjuntos.

```
pares = {x for x in range(1, 11) if x % 2 == 0}
print(pares)
```

Explicación:
Filtramos los números del 1 al 10 que sean divisibles por 2.

Comandos:

- `{expresion for var in iterable if condicion}` → comprensión de conjuntos con filtro.
- `%` → operador módulo (resto).

Ejercicio 32

Enunciado: Encuentra la unión de tres conjuntos `{1,2}`, `{2,3}`, `{3,4}`.

```
a = {1, 2}
b = {2, 3}
c = {3, 4}
union = a | b | c
print(union)
```

Explicación:
Podemos unir más de dos conjuntos con `|`.

Comandos:

- `|` → operador de unión.

Ejercicio 33

Enunciado: Encuentra los elementos comunes en tres conjuntos `{1,2,3}`, `{2,3,4}`, `{3,4,5}`.

```
a = {1, 2, 3}
b = {2, 3, 4}
c = {3, 4, 5}
interseccion = a & b & c
print(interseccion)
```

Explicación:
La intersección de varios conjuntos muestra los elementos comunes a todos.

Comandos:

- `&` → operador de intersección.

Ejercicio 34

Enunciado: Crea un conjunto con los múltiplos de 3 menores que 20.

```
multiplos = {x for x in range(20) if x % 3 == 0}
print(multiplos)
```

Explicación:
Usamos comprensión de conjuntos para generar múltiplos de 3.

Comandos:

- `range(20)` → números del 0 al 19.
- `%` → módulo.

Ejercicio 35

Enunciado: Dado el conjunto `{1,2,3,4,5}`, crea otro con sus elementos elevados al cubo.

```
a = {1, 2, 3, 4, 5}
cubos = {x**3 for x in a}
print(cubos)
```

Explicación:
Usamos comprensión de conjuntos aplicando `x**3` a cada elemento.

Comandos:

- `x**3` → potencia al cubo.

Ejercicio 36

Enunciado: Une dos listas eliminando duplicados con conjuntos.
Listas: `[1,2,2,3]` y `[3,4,4,5]`.

```
lista1 = [1, 2, 2, 3]
lista2 = [3, 4, 4, 5]
union = set(lista1) | set(lista2)
print(list(union))
```

Explicación:
Convertimos listas a conjuntos, unimos y volvemos a lista.

Comandos:

- `set(lista)` → elimina duplicados.
- `|` → unión.

Ejercicio 37

Enunciado: Crea un conjunto con los números impares de 1 a 15.

```
impares = {x for x in range(1, 16) if x % 2 != 0}
print(impares)
```

Explicación:
Filtramos números no divisibles por 2.

Comandos:

- `% 2 != 0` → condición de imparidad.

Ejercicio 38

Enunciado: Comprueba si dos listas contienen exactamente los mismos elementos ignorando duplicados.
Listas: `[1,2,2,3]` y `[3,1,2]`.

```
lista1 = [1, 2, 2, 3]

lista2 = [3, 1, 2]

print(set(lista1) == set(lista2))
```

Explicación:
Al comparar conjuntos, el orden y duplicados no importan.

Comandos:

- `==` → compara igualdad.

Ejercicio 39

Enunciado: Calcula la cantidad de elementos únicos de la lista `[1,2,2,3,3,4,5]`.

```
lista = [1, 2, 2, 3, 3, 4, 5]

unicos = set(lista)

print(len(unicos))
```

Explicación:
Con `set()` eliminamos duplicados y con `len()` contamos.

Comandos:

- `set(lista)` → elimina repetidos.
- `len()` → cuenta elementos.

Ejercicio 40

Enunciado: Dado el conjunto `{1,2,3,4,5}`, elimina los números pares.

```
conjunto = {1, 2, 3, 4, 5}
conjunto = {x for x in conjunto if x % 2 != 0}
print(conjunto)
```

Explicación:
Filtramos los impares con comprensión de conjuntos.

Comandos:

- `if x % 2 != 0` → condición para filtrar impares.

Ejercicio 41

Enunciado: Verifica si todos los elementos de `{1,2,3}` están dentro de `{0,1,2,3,4,5}` usando `<=`.

```
a = {1, 2, 3}
b = {0, 1, 2, 3, 4, 5}
print(a <= b)
```

Explicación:
El operador `<=` comprueba si `a` es subconjunto de `b`.

Comandos:

- `<=` → operador de subconjunto.

Ejercicio 42

Enunciado: Verifica si `{1,2,3,4,5}` contiene a `{2,3}` usando `>=`.

```
a = {1, 2, 3, 4, 5}
b = {2, 3}
print(a >= b)
```

Explicación:
El operador `>=` comprueba si `a` es superconjunto de `b`.

Comandos:

- `>=` → operador de superconjunto.

Ejercicio 43

Enunciado: Elimina los elementos de `{1,2,3,4,5}` que también estén en `{2,4}` usando diferencia.

```
a = {1, 2, 3, 4, 5}
b = {2, 4}
a -= b
print(a)
```

Explicación:
El operador `-=` elimina los elementos comunes directamente del conjunto.

Comandos:

- `-=` → diferencia en asignación.

Ejercicio 44

Enunciado: Encuentra los elementos únicos entre dos listas usando diferencia simétrica.
Listas: `[1,2,3]` y `[3,4,5]`.

```
lista1 = [1, 2, 3]

lista2 = [3, 4, 5]

resultado = set(lista1) ^ set(lista2)

print(resultado)
```

Explicación:
El operador `^` devuelve elementos exclusivos de cada lista.

Comandos:

- `^` → diferencia simétrica.

Ejercicio 45

Enunciado: Convierte un texto en conjunto de palabras únicas.
Texto: `"python es divertido y python es útil"`.

```
texto = "python es divertido y python es útil"

palabras_unicas = set(texto.split())

print(palabras_unicas)
```

Explicación:
Dividimos el texto en palabras con `.split()` y lo pasamos a conjunto.

Comandos:

- `.split()` → separa palabras.
- `set(lista)` → elimina duplicados.

Ejercicio 46

Enunciado: Usa `frozenset` para crear un conjunto inmutable con `{1,2,3}`.

```
conjunto_inmutable = frozenset({1, 2, 3})

print(conjunto_inmutable)
```

Explicación:
Un `frozenset` es como un `set`, pero no se puede modificar.

Comandos:

- `frozenset()` → conjunto inmutable.

Ejercicio 47

Enunciado: Encuentra los caracteres únicos de dos cadenas y únelos.
Cadenas: `"hola"` y `"mundo"`.

```
a = set("hola")
b = set("mundo")
union = a | b
print(union)
```

Explicación:
Convertimos cadenas en conjuntos de letras y unimos con `|`.

Comandos:

- `set(cadena)` → convierte string a conjunto de caracteres.
- `|` → unión.

Ejercicio 48

Enunciado: Encuentra las letras comunes entre `"python"` y `"java"`.

```
a = set("python")
b = set("java")
comunes = a & b
print(comunes)
```

Explicación:
Usamos la intersección & para obtener caracteres en común.

Comandos:

- `&` → intersección.

Ejercicio 49

Enunciado: Crea un conjunto con los divisores de 12.

```
divisores = {x for x in range(1, 13) if 12 % x == 0}
print(divisores)
```

Explicación:
Usamos comprensión de conjuntos verificando qué números dividen a 12.

Comandos:

- `if 12 % x == 0` → condición de divisor.

Ejercicio 50

Enunciado: Usa conjuntos para verificar si una palabra es un isograma (sin letras repetidas). Palabra: `"murcielago"`.

```
palabra = "murcielago"
print(len(set(palabra)) == len(palabra))
```

Explicación:
Si el número de letras únicas es igual al total, no hay repetidas.

Comandos:

- `set(palabra)` → letras únicas.
- `len()` → cuenta elementos.
- `==` → comparación de igualdad.

MANUAL DETALLADO DE EJERCICIOS (1-50): ESTRUCTURA DE DATOS TUPLAS

Manipulación Básica y Definición (Ejercicios 1 - 10)

Ejercicio	Enunciado	Explicación Minuciosa	Comandos y Controles
1	Crea una tupla con los números del 1 al 5 e imprímela.	Una tupla se define utilizando paréntesis `()` y su contenido se muestra en la consola utilizando la función `print()`.	**Definición:** `()` define una tupla. **Visualización:** `print()` muestra el contenido.
2	Crea una tupla con un solo elemento "Python" y comprueba su tipo.	Para que se interprete como una tupla y no como un *string* simple, una tupla de un solo elemento **necesita llevar una coma final**.	**Tupla unitaria:** `("elemento",)` (requiere la coma). **Verificación de tipo:** `type()` devuelve el tipo de dato.
3	Accede al primer y último elemento de la tupla (10,20,30,40,50).	Los elementos de una tupla se acceden mediante **índices**. La indexación comienza en 0 para el primer elemento, y se usa -1 para el último elemento.	**Primer elemento:** `tupla`. **Último elemento:** `tupla[-1]`.
4	Recorre con un `for` la tupla ("rojo","verde","azul").	Es posible iterar sobre cada elemento de una tupla mediante un **bucle `for`**.	**Iteración:** `for var in tupla:` recorre los elementos.
5	Comprueba si "verde" está en la tupla ("rojo","verde","azul").	Se utiliza el operador **`in`** para verificar si un valor específico existe o está contenido dentro de la tupla.	**Verificación de existencia:** `"elemento" in tupla` devuelve `True` o `False`.
6	Une dos tuplas (1,2,3) y (4,5,6) en una sola.	La unión, o **concatenación**, de dos tuplas se logra utilizando el operador `+`.	**Concatenación:** `+` une dos tuplas.
7	Repite la tupla (1,2) tres veces.	El operador de multiplicación **`*`** permite repetir los elementos de una tupla un número *n* de veces.	**Repetición:** `tupla * n` repite la tupla *n* veces.
8	Calcula la longitud de la tupla (10,20,30,40).	La función **`len()`** devuelve la cantidad total de elementos que contiene la tupla.	**Longitud:** `len(tupla)` devuelve la longitud.
9	Convierte una lista en tupla.	Se emplea el constructor **`tuple()`** para transformar un objeto iterable (como una lista o un *string*) en una tupla.	**Conversión:** `tuple(iterable)` crea una tupla.
10	Convierte la tupla (1,2,3) en lista.	La transformación inversa se realiza utilizando la función **`list()`**.	**Conversión:** `list(tupla)` convierte en lista.

Métodos, Índices y Slicing (Ejercicios 11 - 20)

Ejercicio	Enunciado	Explicación Minuciosa	Comandos y Controles
11	Encuentra el número de veces que aparece el número 3 en la tupla (1,2,3,3,4,3,5).	El método **`.count()`** se utiliza para contar cuántas veces aparece un valor específico dentro de la tupla.	**Conteo de ocurrencias:** `tupla.count(valor)`.
12	Encuentra el índice de la primera aparición del número 20 en la tupla (10,20,30,20,40).	El método **`.index(valor)`** devuelve la posición (índice) de la primera vez que se encuentra el elemento buscado.	**Búsqueda de índice:** `tupla.index(valor)`.
13	Desempaqueta la tupla (1,2,3) en tres variables.	Se pueden asignar directamente todos los valores de la tupla a variables individuales en una sola línea. Este proceso se llama **desempaquetado**.	**Desempaquetado:** `a, b, c = tupla`.
14	Intercambia los valores de dos variables usando una tupla.	Python permite intercambiar los valores de dos variables de forma eficiente mediante una **asignación múltiple** (que internamente utiliza una construcción temporal de tupla).	**Intercambio:** `x, y = y, x`.
15	Crea una tupla con los caracteres de la palabra "python".	El constructor **`tuple()`** transforma un *string* en una tupla, donde cada carácter del *string* se convierte en un elemento de la tupla.	**Conversión de *string*:** `tuple("texto")`.
16	Accede a un rango de elementos de la tupla (10,20,30,40,50) del índice 1 al 3.	Se utiliza el **slicing** mediante la sintaxis `[inicio:fin]`. Es crucial recordar que el índice de `fin` no se incluye en el resultado.	**Slicing:** `tupla[1:4]` obtiene elementos desde la posición 1 hasta la 3 (el 4 es excluido).
17	Copia la tupla (1,2,3) en otra variable y muéstrala.	Se puede realizar una copia completa de la tupla utilizando un **slicing completo** (`[:]`).	**Copia:** `tupla[:]` crea una copia de la tupla.
18	Verifica si la tupla (1,2,3) es igual a (1,2,3).	El operador de igualdad **`==`** compara si dos tuplas contienen los mismos valores y en el mismo orden.	**Comparación:** `t1 == t2`.
19	Ordena una tupla (5,2,8,1,3) en una lista.	La función **`sorted()`** recibe la tupla, ordena sus elementos, y devuelve el resultado como una **lista**.	**Ordenamiento:** `sorted(tupla)`.
20	Obtén el valor máximo y mínimo de la tupla (10,20,5,30,15).	Las funciones integradas **`max()`** y **`min()`** encuentran el valor más grande y más pequeño, respectivamente.	**Máximo:** `max(tupla)`. **Mínimo:** `min(tupla)`.

Funciones Agregadas y Tuplas Anidadas (Ejercicios 21 - 30)

Ejercicio	Enunciado	Explicación Minuciosa	Comandos y Controles

21	Suma todos los elementos de la tupla (1,2,3,4,5).	La función **`sum()`** devuelve la suma total de todos los elementos numéricos contenidos en la tupla.	**Suma total:** `sum(tupla)`.
22	Crea una tupla de tuplas con coordenadas 2D: ((1,2),(3,4),(5,6)) y recórrela.	Al iterar sobre una tupla de tuplas (o tuplas anidadas), se puede realizar un **desempaquetado directamente en el bucle `for`**, asignando los valores de la sub-tupla a variables (`x, y`).	**Desempaquetado en bucle:** `for x, y in coordenadas:`. **Impresión con variables:** `f""` (f-string).
23	Comprueba si todas las letras de la tupla ("a","b","c") son diferentes.	Se utiliza un `set()` para **eliminar automáticamente los elementos duplicados**. Si la longitud de la tupla original es igual a la longitud del *set* resultante, significa que no había elementos repetidos.	**Eliminación de duplicados:** `set(tupla)`. **Obtención de longitud:** `len()`.
24	Convierte la tupla (1,2,3,4,5) en un *string* separado por comas.	Primero se usa **`map(str, tupla)`** para transformar cada número en un *string*, y luego **`",".join()`** une todos los elementos utilizando la coma como separador.	**Transformación de tipo:** `map(función, iterable)`. **Unión:** `"sep".join(lista)`.
25	Crea una tupla con los números pares del 2 al 10.	La función **`range(inicio, fin, paso)`** genera la secuencia numérica (2, 4, 6, 8, 10), y luego se utiliza el constructor **`tuple()`** para convertir esta secuencia en una tupla.	**Generación de secuencia:** `range(2,11,2)`. **Conversión:** `tuple()`.
26	Encuentra la longitud de cada palabra en la tupla ("python","java","c").	Se utiliza una **tupla por comprensión** (`()`), aplicando la función `len(palabra)` a cada elemento.	**Tupla por comprensión:** `(expresión for var in iterable)`. **Longitud:** `len(cadena)`.
27	Invierte la tupla (1,2,3,4,5).	El **slicing con paso negativo `[::-1]`** es el control utilizado para devolver los elementos de la tupla en orden inverso.	**Inversión de orden:** `tupla[::-1]`.
28	Encuentra los elementos comunes entre (1,2,3,4) y (3,4,5,6).	Se convierten ambas tuplas a **conjuntos (`set`)** y se utiliza el operador de **intersección (&)** para encontrar los elementos que están presentes en ambas.	**Conversión:** `set(tupla)`. **Intersección:** &.
29	Convierte la tupla (1,2,3,4,5) en un diccionario con índices como claves.	La función **`enumerate()`** genera pares de (índice, valor), y luego la función **`dict()`** toma estos pares para crear el diccionario.	**Generación de índice/valor:** `enumerate(tupla)`. **Creación de diccionario:** `dict()`.
30	Calcula la media de los valores en la tupla (10,20,30,40,50).	La media se calcula dividiendo la suma total de los elementos (`sum(tupla)`) entre la cantidad	**Suma:** `sum(tupla)`. **Cantidad de elementos:** `len(tupla)`.

de elementos (`len(tupla)`).

Comprensiones, Filtrado y Operaciones Avanzadas (Ejercicios 31 - 40)

Ejercicio	Enunciado	Explicación Minuciosa	Comandos y Controles
31	Convierte la tupla (1,2,3,4,5) en un diccionario donde cada número sea la clave y su cuadrado el valor.	Se emplea una **comprensión de diccionario** para iterar sobre la tupla y definir la clave (`x`) y el valor (`x**2`) en una sola línea.	**Comprensión de diccionario:** `{clave: valor for x in iterable}`. **Exponenciación:** `x**2` (eleva al cuadrado).
32	Extrae los números pares de la tupla (1,2,3,4,5,6,7,8,9,10).	Se usa una **comprensión de tupla** con una **condición de filtrado** (`if x % 2 == 0`) para seleccionar solo los números que son pares.	**Condición par:** `x % 2 == 0` (el módulo es cero). **Tupla filtrada:** `(x for x in tupla if condición)`.
33	Genera una tupla con los primeros 10 números cuadrados.	Se combina la generación de números del 1 al 10 usando **`range(1, 11)`** con una **comprensión de tupla** para elevar cada número al cuadrado (`x**2`).	**Generación:** `range(inicio,fin)`. **Potencia:** `x**2`.
34	Encuentra el elemento más largo en la tupla ("python","java","c","javas cript").	Se utiliza la función **`max()`** junto con el argumento **`key=len`**. Esto le indica a `max()` que, en lugar de comparar los elementos directamente, debe compararlos según la longitud que devuelva la función `len()`.	**Máximo por función:** `max(iterable, key=función)`. **Función de longitud:** `len(cadena)`.
35	Combina dos tuplas (1,2,3) y ("a","b","c") en pares.	La función **`zip(t1, t2)`** se utiliza para emparejar elementos de ambas tuplas según su posición. El resultado se convierte en tupla usando `tuple()`.	**Emparejamiento:** `zip(t1,t2)`. **Conversión final:** `tuple(zip(...))`.
36	Encuentra el producto de todos los números en la tupla (1,2,3,4,5).	Se requiere importar la función **`reduce`** de `functools`. `reduce()` aplica una **función lambda** (`x*y`) de forma acumulativa a todos los elementos.	**Importación:** `from functools import reduce`. **Función anónima:** `lambda x,y: x*y`. **Reducción:** `reduce(función, iterable)`.
37	Genera una tupla con los números primos del 1 al 20.	Se necesita definir previamente una **función `def`** (`es_primo`) para verificar la primalidad. Luego, se usa una	**Definición de función:** `def`. **Módulo:** `n % i` (para divisibilidad). **Rango:** `range(1,21)`.

		comprensión de tupla sobre el rango de números, filtrando aquellos que cumplen la condición de ser primos.	
38	Elimina los duplicados de la tupla (1,2,2,3,4,4,5).	Para eliminar duplicados, se convierte temporalmente la tupla a un **conjunto (`set`)**, que garantiza la unicidad, y luego se reconvierte a tupla usando **`tuple()`**.	**Set sin repetidos:** `set(tupla)`. **Reconversión:** `tuple(set(...))`.
39	Encuentra la diferencia entre la tupla (1,2,3,4,5) y (4,5,6,7).	Se convierten ambas tuplas a conjuntos y se utiliza el operador de **resta de conjuntos (`-`)** para obtener los elementos que están en la primera tupla pero no en la segunda.	**Diferencia de conjuntos:** `set(t1) - set(t2)`.
40	Verifica si dos tuplas (1,2,3) y (3,2,1) contienen los mismos elementos sin importar el orden.	Al convertir ambas tuplas a **conjuntos (`set`)**, el orden se elimina. Si los conjuntos resultantes son iguales (==), significa que las tuplas contienen los mismos elementos, independientemente de su posición inicial.	**Eliminación de orden:** `set(tupla)`. **Comparación:** ==.

Filtrado, Inmutabilidad y Conversiones Específicas (Ejercicios 41 - 50)

Ejercicio	Enunciado	Explicación Minuciosa	Comandos y Controles
41	Cuenta cuántos elementos menores que 5 hay en la tupla (1,4,6,2,8,3).	Se utiliza la función **`sum()`** aplicada a un **generador**. El generador produce un valor de 1 por cada elemento que cumple la condición (`x < 5`), y `sum()` los acumula.	**Generador condicional:** `(1 for x in tupla if x < 5)`. **Suma de generador:** `sum()`.
42	Reemplaza los elementos (1,2,3) por (4,5,6) creando una nueva tupla.	Es fundamental recordar que las tuplas son **inmutables**; por lo tanto, no se modifican, sino que se crea una nueva tupla con los valores deseados.	**Definición:** `()` define la tupla. **Visualización:** `print()`.
43	Divide la tupla (10,20,30,40,50) en dos partes iguales.	Se calcula la mitad de la longitud usando **división entera (`//`)** y luego se usa el **slicing** para obtener la primera parte (`[:mitad]`) y la segunda parte (`[mitad:]`).	**Mitad de longitud:** `len(tupla)//2`. **Primeros elementos:** `tupla[:mitad]`. **Elementos restantes:** `tupla[mitad:]`.
44	Convierte la tupla (1,2,3) en una cadena	Similar al Ejercicio 24, se usa **`map(str, tupla)`** para	**Conversión a *string*:** `map(str, tupla)`.

	"1-2-3".	convertir a *strings*, pero el método **`.join(...)`** utiliza el guion (`-`) como separador.	**Unión con guion:** `"-".join(...)`.
45	Encuentra el índice del valor 40 en la tupla (10,20,30,40,50).	El método **`tupla.index(valor)`** devuelve la posición de la primera aparición del valor especificado.	**Búsqueda de índice:** `tupla.index(valor)`.
46	Crea una tupla de 5 elementos todos iguales a 0.	Se utiliza el operador de **repetición (`*`)** sobre una tupla de un solo elemento (que debe llevar la coma para ser reconocida como tupla).	**Repetición con tupla unitaria:** `(0,) * 5`.
47	Intercambia los valores de una tupla (1,2,3) con otra (4,5,6).	Se logra mediante una **asignación múltiple** que intercambia las referencias de las variables `t1` y `t2`.	**Intercambio de variables:** `t1, t2 = t2, t1`.
48	Crea una tupla con números del 1 al 10 y filtra los múltiplos de 3.	Se genera el rango de números con **`range(1, 11)`** y luego se aplica una comprensión de tupla con la condición de **múltiplo de 3** (`x % 3 == 0`).	**Rango:** `range(1,11)`. **Condición:** `x % 3 == 0`.
49	Convierte la tupla (1,2,3,4,5) en un conjunto para eliminar duplicados (aunque no haya).	Se usa el constructor **`set()`** para transformar la tupla en un conjunto, cuyo objetivo principal es garantizar la unicidad de los elementos.	**Conversión a conjunto:** `set(tupla)`.
50	Verifica si una tupla contiene algún número negativo (1,2,-3,4).	La función **`any(iterable)`** comprueba si al menos un elemento de un iterable (en este caso, un generador) cumple con la condición de ser `True`. Se aplica una condición de verificación de negatividad (`x < 0`).	**Verificación de existencia:** `any(iterable)`. **Condición de negativo:** `x < 0`.

Ejercicio 1

Enunciado: Crea una tupla con los números del 1 al 5 e imprímela.

```
tupla = (1, 2, 3, 4, 5)
print(tupla)
```

Explicación:
Una tupla se define con paréntesis `()` y se imprime con `print()`.

Comandos:

- `()` → define una tupla.
- `print()` → muestra el contenido.

Ejercicio 2

Enunciado: Crea una tupla con un solo elemento `"Python"` y comprueba su tipo.

```
tupla = ("Python",)
print(tupla)
print(type(tupla))
```

Explicación:
Una tupla de un solo elemento necesita una coma final, de lo contrario se interpreta como string.

Comandos:

- `("elemento",)` → tupla de un elemento.
- `type()` → devuelve el tipo de dato.

Ejercicio 3

Enunciado: Accede al primer y último elemento de la tupla `(10,20,30,40,50)`.

```
tupla = (10, 20, 30, 40, 50)
print(tupla[0])   # primer elemento
print(tupla[-1])  # último elemento
```

Explicación:
Se accede a los elementos por índice, comenzando en 0.

Comandos:

- `tupla[0]` → primer elemento.
- `tupla[-1]` → último elemento.

Ejercicio 4

Enunciado: Recorre con un `for` la tupla `("rojo","verde","azul")`.

```
colores = ("rojo", "verde", "azul")
for c in colores:
    print(c)
```

Explicación:
Podemos iterar sobre cada elemento de una tupla con un bucle `for`.

Comandos:

- `for var in tupla:` → recorre los elementos.

Ejercicio 5

Enunciado: Comprueba si `"verde"` está en la tupla `("rojo","verde","azul")`.

```
colores = ("rojo", "verde", "azul")
print("verde" in colores)
```

Explicación:
El operador `in` verifica si un valor existe en la tupla.

Comandos:

- `"elemento" in tupla` → devuelve `True` o `False`.

Ejercicio 6

Enunciado: Une dos tuplas `(1,2,3)` y `(4,5,6)` en una sola.

```
a = (1, 2, 3)
b = (4, 5, 6)
c = a + b
print(c)
```

Explicación:
El operador `+` concatena dos tuplas.

Comandos:

- `+` → concatenación de tuplas.

Ejercicio 7

Enunciado: Repite la tupla `(1,2)` tres veces.

```
tupla = (1, 2) * 3
print(tupla)
```

Explicación:
El operador `*` repite los elementos de la tupla.

Comandos:

- `tupla * n` → repite `n` veces.

Ejercicio 8

Enunciado: Calcula la longitud de la tupla `(10,20,30,40)`.

```
tupla = (10, 20, 30, 40)
print(len(tupla))
```

Explicación:
La función `len()` devuelve la cantidad de elementos.

Comandos:

- `len(tupla)` → devuelve longitud.

Ejercicio 9

Enunciado: Convierte una lista `[1,2,3]` en tupla.

```
lista = [1, 2, 3]
tupla = tuple(lista)
print(tupla)
```

Explicación:
Usamos el constructor `tuple()` para transformar una lista en tupla.

Comandos:

- `tuple(iterable)` → crea una tupla a partir de una lista o string.

Ejercicio 10

Enunciado: Convierte la tupla `(1,2,3)` en lista.

```
tupla = (1, 2, 3)
lista = list(tupla)
print(lista)
```

Explicación:
Podemos transformar una tupla en lista con `list()`.

Comandos:

- `list(tupla)` → convierte en lista.

Ejercicio 11

Enunciado: Encuentra el número de veces que aparece el número `3` en la tupla `(1,2,3,3,4,3,5)`.

```
tupla = (1, 2, 3, 3, 4, 3, 5)
print(tupla.count(3))
```

Explicación:
El método `.count()` devuelve cuántas veces aparece un valor dentro de la tupla.

Comandos:

- `tupla.count(valor)` → cuenta ocurrencias.

Ejercicio 12

Enunciado: Encuentra el índice de la primera aparición del número `20` en la tupla `(10,20,30,20,40)`.

```
tupla = (10, 20, 30, 20, 40)
print(tupla.index(20))
```

Explicación:
`.index(valor)` devuelve la posición de la primera vez que aparece el elemento.

Comandos:

- `tupla.index(valor)` → índice del primer elemento encontrado.

Ejercicio 13

Enunciado: Desempaqueta la tupla `(1,2,3)` en tres variables.

```
tupla = (1, 2, 3)
a, b, c = tupla
print(a, b, c)
```

Explicación:
Podemos asignar directamente cada valor de la tupla a distintas variables.

Comandos:

- `a, b, c = tupla` → desempaquetado de tupla.

Ejercicio 14

Enunciado: Intercambia los valores de dos variables usando una tupla.

```
x = 10
y = 20
x, y = y, x
print(x, y)
```

Explicación:
En Python se pueden intercambiar variables con una asignación múltiple usando tuplas.

Comandos:

- `x, y = y, x` → intercambio de valores.

Ejercicio 15

Enunciado: Crea una tupla con los caracteres de la palabra `"python"`.

```
palabra = "python"
tupla = tuple(palabra)
print(tupla)
```

Explicación:
El constructor `tuple()` transforma un string en una tupla de caracteres.

Comandos:

- `tuple("texto")` → convierte en tupla de caracteres.

Ejercicio 16

Enunciado: Accede a un rango de elementos de la tupla `(10,20,30,40,50)` del índice 1 al 3.

```
tupla = (10, 20, 30, 40, 50)
print(tupla[1:4])
```

Explicación:
Se usa el slicing `[inicio:fin]`, donde `fin` no se incluye.

Comandos:

- `tupla[1:4]` → obtiene los elementos desde posición 1 hasta 3.

Ejercicio 17

Enunciado: Copia la tupla `(1,2,3)` en otra variable y muéstrala.

```
tupla = (1, 2, 3)
copia = tupla[:]
print(copia)
```

Explicación:
Hacemos una copia con slicing completo.

Comandos:

- `tupla[:]` → crea copia de la tupla.

Ejercicio 18

Enunciado: Verifica si la tupla `(1,2,3)` es igual a `(1,2,3)`.

```
t1 = (1, 2, 3)
t2 = (1, 2, 3)
print(t1 == t2)
```

Explicación:
El operador `==` compara si dos tuplas tienen los mismos valores.

Comandos:

- `t1 == t2` → comparación de igualdad.

Ejercicio 19

Enunciado: Ordena una tupla `(5,2,8,1,3)` en una lista.

```
tupla = (5, 2, 8, 1, 3)
lista = sorted(tupla)
print(lista)
```

Explicación:
`sorted()` devuelve una lista ordenada de los elementos.

Comandos:

- `sorted(tupla)` → ordena y devuelve lista.

Ejercicio 20

Enunciado: Obtén el valor máximo y mínimo de la tupla `(10,20,5,30,15)`.

```
tupla = (10, 20, 5, 30, 15)
print(max(tupla))
print(min(tupla))
```

Explicación:
Las funciones `max()` y `min()` encuentran el mayor y menor valor.

Comandos:

- `max(tupla)` → máximo valor.
- `min(tupla)` → mínimo valor.

Ejercicio 21

Enunciado: Suma todos los elementos de la tupla `(1,2,3,4,5)`.

```
tupla = (1, 2, 3, 4, 5)
print(sum(tupla))
```

Explicación:
La función `sum()` devuelve la suma de todos los elementos numéricos de la tupla.

Comandos:

- `sum(tupla)` → suma de valores de la tupla.

Ejercicio 22

Enunciado: Crea una tupla de tuplas con coordenadas 2D: `((1,2),(3,4),(5,6))` y recórrela.

```
coordenadas = ((1, 2), (3, 4), (5, 6))
for x, y in coordenadas:
    print(f"x={x}, y={y}")
```

Explicación:
Se recorre la tupla principal y se desempaqueta cada sub-tupla en dos variables.

Comandos:

- `for x, y in coordenadas:` → desempaquetado en el bucle.
- `f""` → f-string para imprimir texto con variables.

Ejercicio 23

Enunciado: Comprueba si todas las letras de la tupla `("a","b","c")` son diferentes.

```
tupla = ("a", "b", "c")
print(len(tupla) == len(set(tupla)))
```

Explicación:
Al convertir la tupla en un `set`, los elementos duplicados desaparecen. Si las longitudes coinciden, no hay repetidos.

Comandos:

- `set(tupla)` → convierte en conjunto eliminando duplicados.
- `len()` → devuelve cantidad de elementos.

Ejercicio 24

Enunciado: Convierte la tupla `(1,2,3,4,5)` en un string separado por comas.

```
tupla = (1, 2, 3, 4, 5)
texto = ",".join(map(str, tupla))
print(texto)
```

Explicación:
Usamos `map(str, tupla)` para transformar cada número en string y `",".join()` para unirlos.

Comandos:

- `map(función, iterable)` → aplica función a cada elemento.
- `"sep".join(lista)` → une elementos con separador.

Ejercicio 25

Enunciado: Crea una tupla con los números pares del 2 al 10.

```
tupla = tuple(range(2, 11, 2))
print(tupla)
```

Explicación:
El constructor `range(inicio, fin, paso)` genera una secuencia numérica y `tuple()` la transforma en tupla.

Comandos:

- `range(2,11,2)` → genera 2,4,6,8,10.
- `tuple()` → convierte en tupla.

Ejercicio 26

Enunciado: Encuentra la longitud de cada palabra en la tupla `("python","java","c")`.

```
tupla = ("python", "java", "c")
longitudes = tuple(len(palabra) for palabra in tupla)
print(longitudes)
```

Explicación:
Se usa una **tupla por comprensión** que obtiene `len()` de cada palabra.

Comandos:

- `(expresión for var in iterable)` → tupla por comprensión.
- `len(cadena)` → longitud de string.

Ejercicio 27

Enunciado: Invierte la tupla `(1,2,3,4,5)`.

```
tupla = (1, 2, 3, 4, 5)
inversa = tupla[::-1]
print(inversa)
```

Explicación:
El slicing con `[::-1]` devuelve los elementos al revés.

Comandos:

- `tupla[::-1]` → invierte el orden.

Ejercicio 28

Enunciado: Encuentra los elementos comunes entre `(1,2,3,4)` y `(3,4,5,6)`.

```
t1 = (1, 2, 3, 4)
t2 = (3, 4, 5, 6)
comunes = tuple(set(t1) & set(t2))
print(comunes)
```

Explicación:
Convertimos las tuplas en conjuntos y usamos & para la intersección.

Comandos:

- `set(tupla)` → convierte en conjunto.
- `&` → intersección de conjuntos.

Ejercicio 29

Enunciado: Convierte la tupla `(1,2,3,4,5)` en un diccionario con índices como claves.

```
tupla = (1, 2, 3, 4, 5)
diccionario = dict(enumerate(tupla))
print(diccionario)
```

Explicación:
`enumerate()` devuelve pares `(índice, valor)` y `dict()` los convierte en diccionario.

Comandos:

- `enumerate(tupla)` → genera índice y valor.
- `dict()` → crea un diccionario.

Ejercicio 30

Enunciado: Calcula la media de los valores en la tupla `(10,20,30,40,50)`.

```
tupla = (10, 20, 30, 40, 50)
media = sum(tupla) / len(tupla)
print(media)
```

Explicación:
La media se obtiene dividiendo la suma de todos los elementos entre la cantidad.

Comandos:

- `sum(tupla)` → suma total.
- `len(tupla)` → número de elementos.

Ejercicio 31

Enunciado: Convierte la tupla `(1,2,3,4,5)` en un diccionario donde cada número sea la clave y su cuadrado el valor.

```
tupla = (1, 2, 3, 4, 5)
diccionario = {x: x**2 for x in tupla}
print(diccionario)
```

Explicación:
Se usa un diccionario por comprensión recorriendo los elementos de la tupla.

Comandos:

- `{clave: valor for x in iterable}` → comprensión de diccionario.
- `x**2` → eleva al cuadrado.

Ejercicio 32

Enunciado: Extrae los números pares de la tupla `(1,2,3,4,5,6,7,8,9,10)`.

```
tupla = (1,2,3,4,5,6,7,8,9,10)
pares = tuple(x for x in tupla if x % 2 == 0)
print(pares)
```

Explicación:
Se usa comprensión de tuplas con una condición.

Comandos:

- `x % 2 == 0` → comprueba si es par.
- `(x for x in tupla if condición)` → genera tupla filtrada.

Ejercicio 33

Enunciado: Genera una tupla con los primeros 10 números cuadrados.

```
tupla = tuple(x**2 for x in range(1, 11))
print(tupla)
```

Explicación:
`range(1,11)` genera del 1 al 10 y se eleva al cuadrado con comprensión.

Comandos:

- `range(inicio,fin)` → genera números.
- `x**2` → potencia al cuadrado.

Ejercicio 34

Enunciado: Encuentra el elemento más largo en la tupla `("python","java","c","javascript")`.

```
tupla = ("python", "java", "c", "javascript")
largo = max(tupla, key=len)
print(largo)
```

Explicación:
Se usa `max()` con el argumento `key=len` para comparar longitudes.

Comandos:

- `max(iterable, key=función)` → devuelve el máximo según función.
- `len(cadena)` → longitud.

Ejercicio 35

Enunciado: Combina dos tuplas `(1,2,3)` y `("a","b","c")` en pares.

```
t1 = (1, 2, 3)
t2 = ("a", "b", "c")
pares = tuple(zip(t1, t2))
print(pares)
```

Explicación:
`zip()` empareja elementos por posición.

Comandos:

- `zip(t1,t2)` → combina en pares.
- `tuple(zip(...))` → convierte en tupla.

Ejercicio 36

Enunciado: Encuentra el producto de todos los números en la tupla `(1,2,3,4,5)`.

```
from functools import reduce

tupla = (1, 2, 3, 4, 5)
producto = reduce(lambda x,y: x*y, tupla)
print(producto)
```

Explicación:
`reduce()` aplica la función `lambda x*y` acumulando los valores.

Comandos:

- `from functools import reduce` → importa `reduce`.
- `lambda x,y: x*y` → función anónima multiplicadora.
- `reduce(función, iterable)` → aplica reducción.

Ejercicio 37

Enunciado: Genera una tupla con los números primos del 1 al 20.

```
def es_primo(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5)+1):
        if n % i == 0:
            return False
    return True

tupla = tuple(x for x in range(1,21) if es_primo(x))
print(tupla)
```

Explicación:
Se define una función `es_primo` y luego se usa en comprensión de tuplas.

Comandos:

- `def` → define función.
- `range(1,21)` → números del 1 al 20.
- `n % i` → módulo para divisibilidad.

Ejercicio 38

Enunciado: Elimina los duplicados de la tupla `(1,2,2,3,4,4,5)`.

```
tupla = (1,2,2,3,4,4,5)
sin_duplicados = tuple(set(tupla))
print(sin_duplicados)
```

Explicación:
Un `set` elimina duplicados, y luego se convierte otra vez a tupla.

Comandos:

- `set(tupla)` → conjunto sin repetidos.
- `tuple(set(...))` → devuelve tupla.

Ejercicio 39

Enunciado: Encuentra la diferencia entre la tupla `(1,2,3,4,5)` y `(4,5,6,7)`.

```
t1 = (1,2,3,4,5)
t2 = (4,5,6,7)
diferencia = tuple(set(t1) - set(t2))
print(diferencia)
```

Explicación:
Usamos la resta de conjuntos para obtener los elementos que están en `t1` y no en `t2`.

Comandos:

- `set(t1) - set(t2)` → diferencia de conjuntos.

Ejercicio 40

Enunciado: Verifica si dos tuplas `(1,2,3)` y `(3,2,1)` contienen los mismos elementos sin importar el orden.

```
t1 = (1,2,3)
t2 = (3,2,1)
print(set(t1) == set(t2))
```

Explicación:
Al convertir en `set`, el orden no importa y se pueden comparar.

Comandos:

- `set(tupla)` → elimina orden y duplicados.
- `==` → comparación de igualdad.

Ejercicio 41

Enunciado: Cuenta cuántos elementos menores que 5 hay en la tupla `(1,4,6,2,8,3)`.

```
tupla = (1,4,6,2,8,3)
menores_5 = sum(1 for x in tupla if x < 5)
print(menores_5)
```

Explicación:
Se recorre la tupla y se suma 1 por cada elemento que cumpla la condición.

Comandos:

- `(1 for x in tupla if x < 5)` → generador de 1 por cada elemento menor que 5.
- `sum()` → suma los valores del generador.

Ejercicio 42

Enunciado: Reemplaza los elementos `(1,2,3)` por `(4,5,6)` creando una nueva tupla.

```
tupla = (1,2,3)
nueva = (4,5,6)
print(nueva)
```

Explicación:
Las tuplas son inmutables; no se pueden modificar directamente, pero se puede crear una nueva.

Comandos:

- `()` → define tupla.
- `print()` → muestra el resultado.

Ejercicio 43

Enunciado: Divide la tupla `(10,20,30,40,50)` en dos partes iguales.

```
tupla = (10,20,30,40,50)
mitad = len(tupla)//2
parte1 = tupla[:mitad]
parte2 = tupla[mitad:]
print(parte1, parte2)
```

Explicación:
Se usa slicing para separar la tupla en dos.

Comandos:

- `len(tupla)//2` → mitad de la longitud.
- `tupla[:mitad]` → primeros elementos.
- `tupla[mitad:]` → elementos restantes.

Ejercicio 44

Enunciado: Convierte la tupla `(1,2,3)` en una cadena `"1-2-3"`.

```
tupla = (1,2,3)
texto = "-".join(map(str, tupla))
print(texto)
```

Explicación:
Se convierte cada elemento a string y se unen con `"-"`.

Comandos:

- `map(str, tupla)` → convierte cada elemento a string.
- `"-".join(...)` → une con guion.

Ejercicio 45

Enunciado: Encuentra el índice del valor `40` en la tupla `(10,20,30,40,50)`.

```
tupla = (10,20,30,40,50)
print(tupla.index(40))
```

Explicación:
`index()` devuelve la posición de la primera aparición del valor.

Comandos:

- `tupla.index(valor)` → índice del elemento.

Ejercicio 46

Enunciado: Crea una tupla de 5 elementos todos iguales a 0.

```
tupla = (0,) * 5
print(tupla)
```

Explicación:
Se usa el operador `*` para repetir elementos en la tupla.

Comandos:

- `(0,) * 5` → tupla con cinco ceros.

Ejercicio 47

Enunciado: Intercambia los valores de una tupla `(1,2,3)` con otra `(4,5,6)`.

```
t1 = (1,2,3)
t2 = (4,5,6)
t1, t2 = t2, t1
print(t1, t2)
```

Explicación:
Se puede hacer intercambio de tuplas con asignación múltiple.

Comandos:

- `t1, t2 = t2, t1` → intercambio de valores.

Ejercicio 48

Enunciado: Crea una tupla con números del 1 al 10 y filtra los múltiplos de 3.

```
tupla = tuple(range(1,11))
multiplos_3 = tuple(x for x in tupla if x % 3 == 0)
print(multiplos_3)
```

Explicación:
Se genera la tupla con `range` y se filtran los múltiplos de 3 con comprensión.

Comandos:

- `range(1,11)` → números del 1 al 10.
- `x % 3 == 0` → condición múltiplo de 3.

Ejercicio 49

Enunciado: Convierte la tupla `(1,2,3,4,5)` en un conjunto para eliminar duplicados (aunque no haya).

```
tupla = (1,2,3,4,5)
conjunto = set(tupla)
print(conjunto)
```

Explicación:
Se transforma en conjunto para eliminar duplicados y mostrar unicidad.

Comandos:

- `set(tupla)` → convierte tupla en conjunto.

Ejercicio 50

Enunciado: Verifica si una tupla contiene algún número negativo `(1,2,-3,4)`.

```
tupla = (1,2,-3,4)
negativo = any(x < 0 for x in tupla)
print(negativo)
```

Explicación:
`any()` devuelve `True` si al menos un elemento cumple la condición.

Comandos:

- `any(iterable)` → comprueba si algún elemento es True.
- `x < 0` → condición para negativo.

MANUAL DETALLADO DE EJERCICIOS: DICCIONARIOS EN PYTHON

Este manual explica minuciosamente el enunciado, la lógica y los comandos esenciales para completar cada ejercicio relacionado con la manipulación de diccionarios.

I. Creación, Acceso y Modificación Básica (Ejercicios 1-10)

Ejercicio	Enunciado y Comando	Explicación Detallada
1	**Crea un diccionario** con los nombres de tres personas y sus edades. `edades = {"Ana": 25, "Luis": 30, "Marta": 22}`	Se define un diccionario utilizando **llaves** `{}` y pares de **`clave: valor`**. **Comandos:** `{clave: valor}` define un diccionario; `print()` muestra el contenido.
2	**Accede a la edad de "Luis"** en el diccionario anterior. `print(edades["Luis"])`	Se utiliza la **clave** (en este caso, `"Luis"`) **entre corchetes** `[]` para obtener el valor asociado a esa clave. **Comandos:** `diccionario[clave]` obtiene el valor.
3	**Agrega una nueva persona** "Carlos" con edad 28 al diccionario. `edades["Carlos"] = 28`	Se asigna un nuevo par **clave-valor** utilizando la clave deseada entre corchetes, lo que añade el elemento si la clave no existe. **Comandos:** `diccionario[clave] = valor` agrega o modifica un elemento.
4	**Elimina a "Marta"** del diccionario. `del edades["Marta"]`	Se utiliza el comando **`del`** seguido del diccionario y la clave entre corchetes para **eliminar** tanto la clave como su valor. **Comandos:** `del diccionario[clave]` elimina el elemento.
5	**Comprueba si "Ana" está** en el diccionario. `print("Ana" in edades)`	Se emplea el operador **`in`** para verificar si una clave específica existe dentro del diccionario. **Comandos:** `"clave" in diccionario` devuelve `True` o `False`.
6	**Obtén todas las claves** del diccionario. `print(edades.keys())`	El método **`.keys()`** devuelve un objeto iterable que contiene todas las claves presentes en el diccionario. **Comandos:** `diccionario.keys()` lista de claves.
7	**Obtén todos los valores** del diccionario. `print(edades.values())`	El método **`.values()`** devuelve todos los valores contenidos en el diccionario. **Comandos:** `diccionario.values()` lista de valores.
8	**Obtén todas las parejas clave-valor** del diccionario. `print(edades.items())`	El método **`.items()`** devuelve tuplas que representan cada par `(clave, valor)` para cada elemento del diccionario. **Comandos:** `diccionario.items()` tuplas clave-valor.
9	**Modifica la edad de "Ana"** a 26. `edades["Ana"] = 26`	Se **sobrescribe** el valor de una clave que ya existe en el diccionario al reasignar un nuevo valor usando la misma sintaxis de acceso por corchetes. **Comandos:** `diccionario[clave] = valor` para actualizar

		el valor.
10	**Recorre el diccionario** e imprime cada clave y su valor. `for nombre, edad in edades.items(): print(f"{nombre} tiene {edad} años")`	Se utiliza un bucle `for` junto con **`.items()`** para iterar simultáneamente sobre los pares `clave, valor` del diccionario. Se usa un **f-string (`f""`)** para dar formato a la salida. **Comandos:** `for clave, valor in diccionario.items():` recorre pares clave-valor; `f""` para formatear la salida.

II. Operaciones de Agregación, Búsqueda y Conversión (Ejercicios 11-20)

Ejercicio	Enunciado y Comando	Explicación Detallada
11	**Suma todas las edades** en el diccionario. `total = sum(edades.values())`	Se obtienen todos los valores numéricos del diccionario usando **`.values()`** y luego se aplica la función **`sum()`** para calcular la suma total. **Comandos:** `diccionario.values()` y `sum()`.
12	**Encuentra la persona con la edad máxima**. `max_persona = max(edades, key=edades.get)`	La función **`max()`** se aplica sobre las claves del diccionario. El argumento **`key=edades.get`** le indica a `max()` que debe usar los valores asociados a esas claves para determinar cuál es la clave con el valor más grande. **Comandos:** `max(diccionario, key=diccionario.get)`.
13	**Invierte el diccionario** para que las edades sean claves y los nombres valores. `invertido = {valor: clave for clave, valor in edades.items()}`	Se crea un nuevo diccionario utilizando una **comprensión de diccionario**, donde se intercambia la posición de la clave y el valor durante la iteración. **Comandos:** `{valor: clave for clave, valor in diccionario.items()}`.
14	**Verifica si "Carlos" está** en el diccionario. `print("Carlos" in edades)`	Similar al Ejercicio 5, se usa el operador **`in`** para comprobar la existencia de una clave. **Comandos:** `"clave" in diccionario` (devuelve `True` si existe, `False` si no).
15	**Elimina la clave "Luis" usando `.pop()`** y muestra su valor. `valor = edades.pop("Luis")`	El método **`.pop(clave)`** elimina la clave especificada del diccionario y, a diferencia de `del`, **devuelve el valor** que estaba asociado a esa clave. **Comandos:** `diccionario.pop(clave)`.
16	**Crea un diccionario vacío** y agrega tres elementos. `dic = {}`	Se inicializa un diccionario vacío usando **`{}`** y los elementos se añaden mediante asignación directa. **Comandos:** `{}` diccionario vacío; `dic[clave] = valor` agrega elemento.
17	**Copia un diccionario usando `.copy()`**. `copia = edades.copy()`	El método **`.copy()`** crea una copia superficial, lo que significa que la nueva copia es independiente de la original para modificaciones de primer nivel. **Comandos:** `diccionario.copy()` devuelve copia del diccionario.
18	**Limpia un diccionario usando `.clear()`**.	El método **`.clear()`** elimina **todos los elementos** dentro del diccionario, dejándolo vacío. **Comandos:**

	`edades.clear()`	`diccionario.clear()` vacía el diccionario.
19	**Combina dos diccionarios** {"a":1,"b":2} y {"b":3,"c":4}. `d1.update(d2)`	El método **`.update(otro_diccionario)`** fusiona el segundo diccionario (`d2`) en el primero (`d1`). Si hay claves duplicadas, los valores del segundo diccionario (d2) **reemplazan** a los del primero (d1). **Comandos:** `diccionario.update(otro_diccionario)` fusiona diccionarios.
20	**Recorre un diccionario e imprime solo las claves**. `for clave in edades: print(clave)`	Cuando se itera directamente sobre un objeto diccionario en un bucle `for`, Python recorre automáticamente solo sus claves. **Comandos:** `for clave in diccionario:` recorre claves.

III. Comprensión, Ordenamiento y Filtrado (Ejercicios 21-30)

Ejercicio	Enunciado y Comando	Explicación Detallada
21	**Obtén todas las claves y valores** usando **`.items()`**. `for clave, valor in dic.items(): print(f"{clave}: {valor}")`	Se usa **`.items()`** para obtener tuplas de pares (clave, valor) y se itera sobre ellas, utilizando un f-string para formatear la salida. **Comandos:** `diccionario.items()` (tuplas clave-valor); `f""` (formateo).
22	**Multiplica todas las edades por 10** y crea uno nuevo. `nuevo = {clave: valor*10 for clave, valor in edades.items()}`	Se emplea la **comprensión de diccionario** para iterar, transformar el valor (`valor*10`) y construir un nuevo diccionario en una sola línea. **Comandos:** `{clave: valor for clave, valor in diccionario.items()}` (comprensión); `valor*10` (operación).
23	**Ordena un diccionario por sus valores**. `ordenado = dict(sorted(edades.items(), key=lambda x: x))`	Se utiliza **`sorted()`** sobre los ítems del diccionario. La función **`key=lambda x: x`** indica que el ordenamiento debe basarse en el segundo elemento de la tupla (el valor), no en la clave. El resultado se convierte de nuevo a diccionario usando **`dict()`**. **Comandos:** `sorted(iterable, key=función)`; `lambda x: x` (función anónima); `dict()`.
24	**Filtra claves con valores mayores a 25**. `filtrado = {k: v for k, v in edades.items() if v > 25}`	Se utiliza una **comprensión de diccionario** añadiendo una cláusula `if` al final para aplicar una condición de filtrado sobre los valores (`if v > 25`). **Comandos:** `if v > 25` (condición); `{k:v for k,v in dic.items() if condición}` (diccionario filtrado).
25	**Intercambia las claves y valores** del diccionario. `invertido = {v: k for k, v in dic.items()}`	Similar al Ejercicio 13, se usa la **comprensión de diccionario** invirtiendo las variables `k` (clave) y `v` (valor). **Comandos:** `{v: k for k, v in dic.items()}` (diccionario por comprensión invertido).
26	**Elimina la clave "b" usando `.popitem()`**. `clave,`	El método **`.popitem()`** elimina y devuelve un par clave-valor. En Python 3.7+ suele eliminar el último par

	`valor = dic.popitem()`	insertado, pero generalmente se considera que elimina un par al azar. **Comandos:** `dic.popitem()` elimina el par y lo devuelve.
27	**Crea un diccionario con claves del 1 al 5 y valores cuadrados**. `dic = {x: x**2 for x in range(1,6)}`	Se usa **comprensión de diccionario** junto con **`range(1, 6)`** (que genera números del 1 al 5) para calcular y asignar el cuadrado (`x**2`) a cada número. **Comandos:** `range(1,6)`; `x**2`; `{x:x**2 for x in ...}` (comprensión).
28	**Copia las claves de un diccionario en una lista**. `claves = list(dic.keys())`	Se obtienen las claves usando **`.keys()`** y luego se utiliza la función **`list()`** para convertir este objeto iterable de claves en una lista estándar. **Comandos:** `dic.keys()`; `list()` convierte iterable en lista.
29	Crea un diccionario donde cada letra del string "python" sea clave y su índice sea valor. `dic = {letra: i for i, letra in enumerate(texto)}`	Se utiliza la función **`enumerate(iterable)`**, que devuelve tanto el índice (`i`) como el valor (`letra`) durante la iteración, dentro de una comprensión de diccionario. **Comandos:** `enumerate(iterable)`; `{letra:i for i, letra in ...}`.
30	**Combina dos diccionarios usando el operador `	`(Python 3.9+)**.` d3 = d1

IV. Operaciones Avanzadas y Lógica (Ejercicios 31-40)

Ejercicio	Enunciado y Comando	Explicación Detallada
31	**Suma solo los valores pares** de un diccionario. `suma_pares = sum(v for v in dic.values() if v % 2 == 0)`	Se usa una **expresión generadora** dentro de la función **`sum()`**. Esta expresión primero obtiene todos los valores (`dic.values()`), los filtra usando la condición de paridad (`v % 2 == 0`), y finalmente los suma. **Comandos:** `dic.values()`; `v % 2 == 0` (verifica paridad); `sum()`.
32	**Multiplica por 2 todos los valores** y actualízalo. `for clave in dic: dic[clave] *= 2`	Se recorren las claves del diccionario. Dentro del bucle, se accede al valor de cada clave y se actualiza in situ utilizando el operador de asignación aumentada **`*=`**. **Comandos:** `for clave in dic:`; `dic[clave] *= 2` (multiplica y actualiza).
33	**Elimina todas las claves con valor menor que 3**. `dic = {k:v for k,v in dic.items() if v >= 3}`	Para eliminar elementos mientras se itera, se crea un **nuevo diccionario filtrado** usando comprensión, incluyendo solo los pares clave-valor que cumplen la condición de tener un valor mayor o igual a 3 (`if v >= 3`). **Comandos:** `{k:v for k,v in dic.items() if condición}`.
34	**Encuentra la clave con el valor mínimo**. `min_clave = min(dic, key=dic.get)`	Similar al Ejercicio 12, se usa la función **`min()`** sobre las claves, indicando con **`key=dic.get`** que la comparación se debe realizar utilizando los valores asociados a esas claves. **Comandos:** `min(dic, key=dic.get)`.
35	**Combina dos diccionarios** y conserva los valores de ambos en listas si hay claves	set(d2): ...`

	repetidas. `for k in set(d1)	
36	**Verifica si algún valor es mayor a 5**. `print(any(v > 5 for v in dic.values()))`	La función **`any()`** devuelve `True` si al menos un elemento dentro del iterable cumple la condición. Se genera el iterable filtrado usando valores y la condición `v > 5`. **Comandos:** `any(iterable)`; `v > 5` (condición).
37	**Verifica si todos los valores son positivos**. `print(all(v > 0 for v in dic.values()))`	La función **`all()`** devuelve `True` solo si *todos* los elementos del iterable cumplen la condición especificada (en este caso, `v > 0`). **Comandos:** `all(iterable)`.
38	**Convierte una lista de tuplas** `[("a",1), ("b",2)]` en diccionario. `dic = dict(lista)`	La función **`dict()`** puede tomar una lista de tuplas (donde cada tupla es un par `(clave, valor)`) y transformarla directamente en un diccionario. **Comandos:** `dict(iterable_de_pares)`.
39	**Copia un diccionario y agrega un nuevo elemento a la copia sin afectar al original**. `copia = dic.copy()`	Se usa **`.copy()`** para crear una copia superficial e independiente. Las modificaciones a la `copia` (como agregar `copia["c"] = 3`) no afectarán al diccionario original. **Comandos:** `dic.copy()` (copia superficial).
40	**Obtén el valor de la clave "d" sin que genere error si no existe, usando `.get()`**. `valor = dic.get("d", 0)`	El método **`.get(clave, valor_por_defecto)`** accede al valor de forma segura. Si la clave no existe, devuelve el `valor_por_defecto` especificado (en este caso, 0), evitando así un error. **Comandos:** `dic.get(clave, valor_por_defecto)` (acceso seguro).

V. Métodos de Iteración, Conversión de Listas y Contadores (Ejercicios 41-50)

Ejercicio	Enunciado y Comando	Explicación Detallada
41	**Suma los valores de un diccionario usando un bucle for**. `for valor in dic.values(): total += valor`	Se utiliza un bucle `for` para iterar explícitamente sobre los valores del diccionario obtenidos mediante **`.values()`**, acumulando la suma en una variable `total` con el operador `+=`. **Comandos:** `for valor in dic.values():`; `+=` (suma acumulativa).
42	**Convierte dos listas** `["a","b","c"]` y `` en un diccionario**. `dic = dict(zip(claves, valores))`	La función **`zip()`** combina elementos de ambas listas por su posición, creando pares `(clave, valor)`. Luego, **`dict()`** convierte esos pares en un diccionario. **Comandos:** `zip(lista1, lista2)` (combina por posición); `dict()` (convierte a diccionario).
43	**Invierte un diccionario** `{"a":1,"b":2}` en `{1:"a",2:"b"}`. `invertido = {v:k for k,v in dic.items()}`	Se utiliza la **comprensión de diccionario** para intercambiar las variables que representan la clave (`k`) y el valor (`v`). **Comandos:** `{v:k for k,v in dic.items()}` (diccionario invertido).
44	**Filtra claves que empiecen con**	Se emplea la comprensión de diccionario y el método

	"**a**". `filtrado = {k:v for k,v in dic.items() if k.startswith("a")}`	de string **`.startswith("a")`** como condición de filtrado (`if`) para verificar que la clave (`k`) comienza con la letra deseada. **Comandos:** `k.startswith("a")` (verifica inicio); `{k:v for k,v in dic.items() if condición}`.
45	**Combina dos diccionarios y suma los valores** de claves repetidas. `resultado[k] = d1.get(k,0) + d2.get(k,0)`	Este método combina claves usando la **unión de conjuntos** (`set(d1)
46	**Crea un diccionario con números del 1 al 5 como claves y sus cubos como valores**. `dic = {x:x**3 for x in range(1,6)}`	Se utiliza la **comprensión de diccionario** y la operación de potenciación `**x**3` para generar los cubos de los números obtenidos de `range(1, 6)`. **Comandos:** `range(1,6)`; `x**3` (cubo de x); `{x:x**3 for x in ...}`.
47	**Recorre un diccionario y muestra solo los valores mayores a 10**. `for valor in dic.values(): if valor > 10: print(valor)`	Se itera sobre los valores obtenidos con **`.values()`** y se aplica una **estructura condicional `if`** dentro del bucle para filtrar la salida. **Comandos:** `for valor in dic.values():`; `if valor > 10:` (condición de filtrado).
48	**Actualiza un diccionario con otro usando `.update()`**. `d1.update(d2)`	Similar al Ejercicio 19, **`.update(otro_dic)`** añade o reemplaza las claves del primer diccionario (`d1`) con los elementos del segundo (`d2`). **Comandos:** `dic.update(otro_dic)` (fusiona diccionarios).
49	**Crea un diccionario con letras de una palabra como claves y su frecuencia como valor**. `dic[letra] = dic.get(letra,0) + 1`	Se utiliza un bucle para recorrer la palabra. Dentro del bucle, se usa **`.get(clave, 0)`** para obtener el contador actual (o 0 si es la primera vez que aparece la letra) y luego se incrementa en 1 (`+= 1`). **Comandos:** `dic.get(clave, 0)` (obtiene valor o 0); `+= 1` (incrementa contador).
50	**Limpia un diccionario y verifica que está vacío**. `dic.clear(); print(len(dic) == 0)`	Se usa el método **`.clear()`** para eliminar todos los elementos. Para verificar que está vacío, se emplea la función **`len(dic)`** para obtener el número de elementos y se comprueba si es igual a cero (`== 0`). **Comandos:** `dic.clear()` (vacía diccionario); `len(dic)` (número de elementos); `== 0` (verifica vacío).

Ejercicio 1

Enunciado: Crea un diccionario con los nombres de tres personas y sus edades.

```
edades = {"Ana": 25, "Luis": 30, "Marta": 22}
print(edades)
```

Explicación:
Se define un diccionario con `{clave: valor}` y se imprime.

Comandos:

- `{clave: valor}` → define un diccionario.
- `print()` → muestra el contenido.

Ejercicio 2

Enunciado: Accede a la edad de "Luis" en el diccionario anterior.

```
edades = {"Ana": 25, "Luis": 30, "Marta": 22}
print(edades["Luis"])
```

Explicación:
Se usa la clave entre corchetes para acceder al valor.

Comandos:

- `diccionario[clave]` → obtiene el valor asociado a la clave.

Ejercicio 3

Enunciado: Agrega una nueva persona `"Carlos"` con edad `28` al diccionario.

```
edades = {"Ana": 25, "Luis": 30, "Marta": 22}
edades["Carlos"] = 28
print(edades)
```

Explicación:
Se asigna un nuevo par clave-valor usando la clave.

Comandos:

- `diccionario[clave] = valor` → agrega o modifica un elemento.

Ejercicio 4

Enunciado: Elimina a `"Marta"` del diccionario.

```
edades = {"Ana": 25, "Luis": 30, "Marta": 22}
del edades["Marta"]
print(edades)
```

Explicación:
`del` elimina la clave y su valor del diccionario.

Comandos:

- `del diccionario[clave]` → elimina el elemento.

Ejercicio 5

Enunciado: Comprueba si `"Ana"` está en el diccionario.

```
edades = {"Ana": 25, "Luis": 30, "Marta": 22}
print("Ana" in edades)
```

Explicación:
El operador `in` verifica si la clave existe en el diccionario.

Comandos:

- `"clave" in diccionario` → devuelve True o False.

Ejercicio 6

Enunciado: Obtén todas las claves del diccionario.

```
edades = {"Ana": 25, "Luis": 30, "Marta": 22}
print(edades.keys())
```

Explicación:
`.keys()` devuelve un objeto iterable con todas las claves.

Comandos:

- `diccionario.keys()` → lista de claves.

Ejercicio 7

Enunciado: Obtén todos los valores del diccionario.

```
edades = {"Ana": 25, "Luis": 30, "Marta": 22}
print(edades.values())
```

Explicación:
`.values()` devuelve todos los valores.

Comandos:

- `diccionario.values()` → lista de valores.

Ejercicio 8

Enunciado: Obtén todas las parejas clave-valor del diccionario.

```
edades = {"Ana": 25, "Luis": 30, "Marta": 22}
print(edades.items())
```

Explicación:
`.items()` devuelve tuplas `(clave, valor)` para cada elemento.

Comandos:

- `diccionario.items()` → tuplas clave-valor.

Ejercicio 9

Enunciado: Modifica la edad de `"Ana"` a `26`.

```
edades = {"Ana": 25, "Luis": 30, "Marta": 22}
edades["Ana"] = 26
print(edades)
```

Explicación:
Se sobrescribe el valor de una clave existente.

Comandos:

- `diccionario[clave] = valor` → actualizar valor.

Ejercicio 10

Enunciado: Recorre el diccionario e imprime cada clave y su valor.

```
edades = {"Ana": 25, "Luis": 30, "Marta": 22}
for nombre, edad in edades.items():
    print(f"{nombre} tiene {edad} años")
```

Explicación:
Se recorre el diccionario con `.items()` para obtener claves y valores.

Comandos:

- `for clave, valor in diccionario.items():` → recorre pares clave-valor.
- `f""` → f-string para formatear salida.

Ejercicio 11

Enunciado: Suma todas las edades en el diccionario `{"Ana":25, "Luis":30, "Marta":22}`.

```
edades = {"Ana":25, "Luis":30, "Marta":22}
total = sum(edades.values())
print(total)
```

Explicación:
Se suman todos los valores del diccionario usando `.values()`.

Comandos:

- `diccionario.values()` → obtiene los valores.
- `sum()` → suma los elementos.

Ejercicio 12

Enunciado: Encuentra la persona con la edad máxima.

```
edades = {"Ana":25, "Luis":30, "Marta":22}
max_persona = max(edades, key=edades.get)
print(max_persona)
```

Explicación:
`max()` con `key=edades.get` devuelve la clave con el valor más grande.

Comandos:

- `max(diccionario, key=diccionario.get)` → devuelve clave con valor máximo.
- `diccionario.get(clave)` → obtiene valor de la clave.

Ejercicio 13

Enunciado: Invierte el diccionario para que las edades sean claves y los nombres valores.

```
edades = {"Ana":25, "Luis":30, "Marta":22}
invertido = {valor: clave for clave, valor in edades.items()}
print(invertido)
```

Explicación:
Se crea un diccionario por comprensión intercambiando clave y valor.

Comandos:

- `{valor: clave for clave, valor in diccionario.items()}` → comprensión de diccionario.
- `.items()` → recorre pares clave-valor.

Ejercicio 14

Enunciado: Verifica si `"Carlos"` está en el diccionario.

```
edades = {"Ana":25, "Luis":30, "Marta":22}
print("Carlos" in edades)
```

Explicación:
`in` comprueba si una clave existe en el diccionario.

Comandos:

- `"clave" in diccionario` → True si existe, False si no.

Ejercicio 15

Enunciado: Elimina la clave `"Luis"` usando `.pop()` y muestra su valor.

```
edades = {"Ana":25, "Luis":30, "Marta":22}
valor = edades.pop("Luis")
print(valor)
print(edades)
```

Explicación:
`.pop(clave)` elimina la clave y devuelve su valor.

Comandos:

- `diccionario.pop(clave)` → elimina clave y devuelve valor.

Ejercicio 16

Enunciado: Crea un diccionario vacío y agrega tres elementos.

```
dic = {}
dic["a"] = 1
dic["b"] = 2
dic["c"] = 3
print(dic)
```

Explicación:
Se define un diccionario vacío `{}` y se agregan elementos con asignación.

Comandos:

- `{}` → diccionario vacío.
- `dic[clave] = valor` → agrega elemento.

Ejercicio 17

Enunciado: Copia un diccionario usando `.copy()`.

```
edades = {"Ana":25, "Luis":30, "Marta":22}
copia = edades.copy()
print(copia)
```

Explicación:
`.copy()` crea una copia superficial del diccionario.

Comandos:

- `diccionario.copy()` → devuelve copia del diccionario.

Ejercicio 18

Enunciado: Limpia un diccionario usando `.clear()`.

```
edades = {"Ana":25, "Luis":30, "Marta":22}
edades.clear()
print(edades)
```

Explicación:
`.clear()` elimina todos los elementos del diccionario.

Comandos:

- `diccionario.clear()` → vacía el diccionario.

Ejercicio 19

Enunciado: Combina dos diccionarios `{"a":1,"b":2}` y `{"b":3,"c":4}`.

```
d1 = {"a":1,"b":2}
d2 = {"b":3,"c":4}
d1.update(d2)
print(d1)
```

Explicación:
`.update()` agrega o reemplaza claves de otro diccionario.

Comandos:

- `diccionario.update(otro_diccionario)` → fusiona diccionarios.

Ejercicio 20

Enunciado: Recorre un diccionario e imprime solo las claves.

```
edades = {"Ana":25, "Luis":30, "Marta":22}
for clave in edades:
    print(clave)
```

Explicación:
Iterar directamente sobre el diccionario recorre sus claves.

Comandos:

- `for clave in diccionario:` → recorre claves.

Ejercicio 21

Enunciado: Obtén todas las claves y valores del diccionario `{"a":1,"b":2,"c":3}` usando `.items()`.

```
dic = {"a":1,"b":2,"c":3}
for clave, valor in dic.items():
    print(f"{clave}: {valor}")
```

Explicación:
`.items()` devuelve pares `(clave, valor)` que se pueden recorrer en un bucle.

Comandos:

- `diccionario.items()` → tuplas clave-valor.
- `for clave, valor in ...` → recorre los pares.
- `f""` → formateo de salida.

Ejercicio 22

Enunciado: Multiplica todas las edades del diccionario `{"Ana":2, "Luis":3}` por 10 y crea uno nuevo.

```
edades = {"Ana":2, "Luis":3}
nuevo = {clave: valor*10 for clave, valor in edades.items()}
print(nuevo)
```

Explicación:
Se usa un **diccionario por comprensión** para transformar los valores.

Comandos:

- `{clave: valor for clave, valor in diccionario.items()}` → comprensión.
- `valor*10` → operación sobre valor.

Ejercicio 23

Enunciado: Ordena un diccionario por sus valores.

```
edades = {"Ana":25, "Luis":30, "Marta":22}
ordenado = dict(sorted(edades.items(), key=lambda x: x[1]))
print(ordenado)
```

Explicación:
`sorted()` con `key=lambda x: x[1]` ordena por valores, luego se convierte en diccionario.

Comandos:

- `sorted(iterable, key=función)` → ordena elementos.
- `lambda x: x[1]` → función anónima que retorna el valor.
- `dict()` → convierte lista de tuplas en diccionario.

Ejercicio 24

Enunciado: Filtra claves con valores mayores a 25.

```
edades = {"Ana":25, "Luis":30, "Marta":22}
filtrado = {k: v for k, v in edades.items() if v > 25}
print(filtrado)
```

Explicación:
Se aplica un **filtro en la comprensión de diccionario**.

Comandos:

- `if v > 25` → condición para filtrar valores.
- `{k:v for k,v in dic.items() if condición}` → diccionario filtrado.

Ejercicio 25

Enunciado: Intercambia las claves y valores del diccionario `{"a":1,"b":2}`.

```
dic = {"a":1,"b":2}
invertido = {v: k for k, v in dic.items()}
print(invertido)
```

Explicación:
Se crea un nuevo diccionario con los pares invertidos.

Comandos:

- `{v: k for k, v in dic.items()}` → diccionario por comprensión invertido.

Ejercicio 26

Enunciado: Elimina la clave `"b"` usando `.popitem()`.

```
dic = {"a":1,"b":2,"c":3}
clave, valor = dic.popitem()
print(clave, valor)
print(dic)
```

Explicación:
`.popitem()` elimina y devuelve un par clave-valor al azar (último en Python 3.7+).

Comandos:

- `dic.popitem()` → elimina último par y lo devuelve.

Ejercicio 27

Enunciado: Crea un diccionario con claves del 1 al 5 y valores cuadrados.

```
dic = {x: x**2 for x in range(1,6)}
print(dic)
```

Explicación:
Se genera un diccionario por comprensión con los cuadrados de los números.

Comandos:

- `range(1,6)` → números del 1 al 5.
- `x**2` → cuadrado del número.
- `{x:x**2 for x in ...}` → diccionario por comprensión.

Ejercicio 28

Enunciado: Copia las claves de un diccionario en una lista.

```
dic = {"a":1,"b":2,"c":3}
claves = list(dic.keys())
print(claves)
```

Explicación:
Se convierten las claves en lista usando `list()`.

Comandos:

- `dic.keys()` → obtiene claves.
- `list()` → convierte iterable en lista.

Ejercicio 29

Enunciado: Crea un diccionario donde cada letra del string `"python"` sea clave y su índice sea valor.

```
texto = "python"
dic = {letra: i for i, letra in enumerate(texto)}
print(dic)
```

Explicación:
`enumerate()` devuelve índice y letra, usado en comprensión de diccionario.

Comandos:

- `enumerate(iterable)` → índice y valor.
- `{letra:i for i, letra in ...}` → diccionario por comprensión.

Ejercicio 30

Enunciado: Combina dos diccionarios usando el operador `|` (Python 3.9+).

```
d1 = {"a":1, "b":2}
d2 = {"b":3, "c":4}
d3 = d1 | d2
print(d3)
```

Explicación:
El operador `|` une diccionarios, reemplazando claves duplicadas con las del segundo.

Comandos:

- `d1 | d2` → combina diccionarios.

Ejercicio 31

Enunciado: Suma solo los valores pares de un diccionario `{"a":2, "b":3, "c":4}`.

```
dic = {"a":2, "b":3, "c":4}
suma_pares = sum(v for v in dic.values() if v % 2 == 0)
print(suma_pares)
```

Explicación:
Se filtran los valores pares con comprensión y luego se suman.

Comandos:

- `dic.values()` → obtiene valores.
- `v % 2 == 0` → verifica si es par.
- `sum()` → suma valores.

Ejercicio 32

Enunciado: Multiplica por 2 todos los valores de un diccionario y actualízalo.

```
dic = {"a":1, "b":2, "c":3}
for clave in dic:
    dic[clave] *= 2
print(dic)
```

Explicación:
Se recorre cada clave y se actualiza su valor.

Comandos:

- `for clave in dic:` → recorre claves.
- `dic[clave] *= 2` → multiplica y actualiza el valor.

Ejercicio 33

Enunciado: Elimina todas las claves con valor menor que 3.

```
dic = {"a":1, "b":4, "c":2}
dic = {k:v for k,v in dic.items() if v >= 3}
print(dic)
```

Explicación:
Se crea un nuevo diccionario filtrando por valor.

Comandos:

- `{k:v for k,v in dic.items() if condición}` → diccionario filtrado.

Ejercicio 34

Enunciado: Encuentra la clave con el valor mínimo.

```
dic = {"a":5, "b":2, "c":9}
min_clave = min(dic, key=dic.get)
print(min_clave)
```

Explicación:
`min()` con `key=dic.get` devuelve la clave con valor mínimo.

Comandos:

- `min(dic, key=dic.get)` → clave con valor mínimo.

Ejercicio 35

Enunciado: Combina dos diccionarios y conserva los valores de ambos en listas si hay claves repetidas.

```
d1 = {"a":1, "b":2}
d2 = {"b":3, "c":4}
combinado = {}
for k in set(d1) | set(d2):
    combinado[k] = []
    if k in d1: combinado[k].append(d1[k])
    if k in d2: combinado[k].append(d2[k])
print(combinado)
```

Explicación:
Se combinan claves usando unión de conjuntos y se almacenan los valores en listas.

Comandos:

- `set(d1) | set(d2)` → unión de claves.
- `append()` → agrega a lista.

Ejercicio 36

Enunciado: Verifica si algún valor es mayor a 5.

```
dic = {"a":3, "b":6, "c":2}
print(any(v > 5 for v in dic.values()))
```

Explicación:
`any()` devuelve True si al menos un valor cumple la condición.

Comandos:

- `any(iterable)` → verifica condición.
- `v > 5` → condición sobre valor.

Ejercicio 37

Enunciado: Verifica si todos los valores son positivos.

```
dic = {"a":3, "b":6, "c":2}
print(all(v > 0 for v in dic.values()))
```

Explicación:
`all()` devuelve True solo si todos cumplen la condición.

Comandos:

- `all(iterable)` → verifica condición para todos los elementos.

Ejercicio 38

Enunciado: Convierte una lista de tuplas `[("a",1),("b",2)]` en diccionario.

```
lista = [("a",1),("b",2)]
dic = dict(lista)
print(dic)
```

Explicación:
`dict()` convierte una lista de pares `(clave, valor)` en diccionario.

Comandos:

- `dict(iterable_de_pares)` → crea diccionario.

Ejercicio 39

Enunciado: Copia un diccionario y agrega un nuevo elemento a la copia sin afectar al original.

```
dic = {"a":1,"b":2}
copia = dic.copy()
copia["c"] = 3
print(dic)
print(copia)
```

Explicación:
`.copy()` crea una copia independiente para modificar sin afectar al original.

Comandos:

- `dic.copy()` → copia superficial.

Ejercicio 40

Enunciado: Obtén el valor de la clave `"d"` sin que genere error si no existe, usando `.get()`.

```
dic = {"a":1,"b":2}
valor = dic.get("d", 0)
print(valor)
```

Explicación:
`.get()` devuelve el valor si existe o un valor por defecto si no.

Comandos:

- `dic.get(clave, valor_por_defecto)` → acceso seguro.

Ejercicio 41

Enunciado: Suma los valores de un diccionario usando un bucle `for`.

```
dic = {"a":1, "b":2, "c":3}
total = 0
for valor in dic.values():
    total += valor
print(total)
```

Explicación:
Se recorre cada valor y se acumula en la variable `total`.

Comandos:

- `for valor in dic.values():` → recorre los valores.
- `+=` → suma acumulativa.

Ejercicio 42

Enunciado: Convierte dos listas `["a","b","c"]` y `[1,2,3]` en un diccionario.

```
claves = ["a","b","c"]
valores = [1,2,3]
dic = dict(zip(claves, valores))
print(dic)
```

Explicación:
`zip()` combina las listas y `dict()` lo convierte en diccionario.

Comandos:

- `zip(lista1, lista2)` → combina elementos por posición.
- `dict()` → convierte en diccionario.

Ejercicio 43

Enunciado: Invierte un diccionario `{"a":1,"b":2}` en `{1:"a",2:"b"}`.

```
dic = {"a":1,"b":2}
invertido = {v:k for k,v in dic.items()}
print(invertido)
```

Explicación:
Se intercambian claves y valores usando comprensión de diccionario.

Comandos:

- `{v:k for k,v in dic.items()}` → diccionario invertido.

Ejercicio 44

Enunciado: Filtra claves que empiecen con `"a"`.

```
dic = {"a1":10,"b2":20,"a3":30}
filtrado = {k:v for k,v in dic.items() if k.startswith("a")}
print(filtrado)
```

Explicación:
`startswith()` verifica si la clave comienza con `"a"`.

Comandos:

- `k.startswith("a")` → True si empieza con "a".
- `{k:v for k,v in dic.items() if condición}` → diccionario filtrado.

Ejercicio 45

Enunciado: Combina dos diccionarios y suma los valores de claves repetidas.

```
d1 = {"a":2, "b":3}
d2 = {"b":4, "c":5}
resultado = {}
for k in set(d1) | set(d2):
    resultado[k] = d1.get(k,0) + d2.get(k,0)
print(resultado)
```

Explicación:
Se suman valores usando `get()` para evitar errores con claves que no existan.

Comandos:

- `set(d1) | set(d2)` → unión de claves.
- `dic.get(clave, 0)` → obtiene valor o 0 si no existe.

Ejercicio 46

Enunciado: Crea un diccionario con números del 1 al 5 como claves y sus cubos como valores.

```
dic = {x:x**3 for x in range(1,6)}
print(dic)
```

Explicación:
Se usa comprensión de diccionario para calcular cubos.

Comandos:

- `range(1,6)` → números del 1 al 5.
- `x**3` → cubo de x.
- `{x:x**3 for x in ...}` → diccionario por comprensión.

Ejercicio 47

Enunciado: Recorre un diccionario y muestra solo los valores mayores a 10.

```
dic = {"a":5,"b":15,"c":20}
for valor in dic.values():
    if valor > 10:
        print(valor)
```

Explicación:
Se filtran los valores mediante condicional dentro del bucle.

Comandos:

- `for valor in dic.values():` → recorre valores.
- `if valor > 10:` → condición de filtrado.

Ejercicio 48

Enunciado: Actualiza un diccionario con otro usando `.update()`.

```
d1 = {"a":1,"b":2}
d2 = {"b":3,"c":4}
d1.update(d2)
print(d1)
```

Explicación:
`update()` añade o reemplaza claves de `d1` con las de `d2`.

Comandos:

- `dic.update(otro_dic)` → fusiona diccionarios.

Ejercicio 49

Enunciado: Crea un diccionario con letras de una palabra como claves y su frecuencia como valor.

```
palabra = "python"
dic = {}
for letra in palabra:
    dic[letra] = dic.get(letra,0) + 1
print(dic)
```

Explicación:
Se cuenta la frecuencia de cada letra usando `get()` para inicializar valores.

Comandos:

- `dic.get(clave, 0)` → obtiene valor o 0 si no existe.
- `+= 1` → incrementa contador.

Ejercicio 50

Enunciado: Limpia un diccionario y verifica que está vacío.

```
dic = {"a":1,"b":2}
dic.clear()
print(dic)
print(len(dic) == 0)
```

Explicación:
`.clear()` elimina todos los elementos y `len()` comprueba que no hay elementos.

Comandos:

- `dic.clear()` → vacía diccionario.
- `len(dic)` → número de elementos.
- `== 0` → verifica si está vacío.

MANUAL DETALLADO DE FUNCIONES EN PYTHON

Bloque 1: Introducción y Operaciones Básicas

Ejercicio 1: Saludo Simple

Enunciado: Crea una función `saludo` que imprima "Hola, mundo". **Solución:**

```
def saludo():
    print("Hola, mundo")
saludo()
```

Explicación Detallada:

1. **Definición:** Se utiliza el comando `def` seguido del nombre de la función (`saludo`) y paréntesis `()` para definir la función.
2. **Acción:** Dentro de la función, el comando `print()` se utiliza para mostrar el texto especificado ("Hola, mundo") en la pantalla.
3. **Ejecución:** Se llama a la función con `saludo()` para que el código contenido en ella se ejecute. **Comandos Usados:**

- `def`: Define una función.
- `print()`: Muestra texto o datos en pantalla.

Ejercicio 2: Suma de Dos Números

Enunciado: Crea una función `suma` que reciba dos números y devuelva su suma. **Solución:**

```
def suma(a, b):
    return a + b
resultado = suma(3, 5)
print(resultado)
```

Explicación Detallada:

1. **Parámetros:** La función `suma` acepta dos valores de entrada, identificados como parámetros `a` y `b`.
2. **Retorno:** El comando `return` se utiliza para devolver el resultado de la expresión `a + b`.
3. **Almacenamiento:** Al llamar a la función con `suma(3, 5)`, el valor devuelto se almacena en una `Variable` llamada `resultado`. **Comandos Usados:**

- `return`: Devuelve un valor de la función.
- `Variables`: Se utilizan para almacenar datos, como el resultado de la función.
- `print()`: Muestra resultados.

Ejercicio 3: Verificar Paridad

Enunciado: Crea una función `es_par` que reciba un número y devuelva `True` si es par, `False` si es impar. **Solución:**

```
def es_par(num):
    if num % 2 == 0:
        return True
    else:
        return False
```

Explicación Detallada:

1. **Módulo:** El operador `%` (módulo) calcula el resto de la división del número entre 2.
2. **Condicional:** El comando `if` verifica si el resto es igual a cero (`num % 2 == 0`).
3. **Flujo:** Si la condición se cumple, devuelve `True`; si no, la estructura `else` se ejecuta y devuelve `False`. **Comandos Usados:**

- `%`: Operador módulo (resto de la división).
- `if-else`: Estructura condicional para verificar una condición y actuar en consecuencia.
- `return`: Devuelve un valor (`True` o `False`).

Ejercicio 4: Encontrar el Mayor

Enunciado: Crea una función `mayor` que reciba dos números y devuelva el mayor. **Solución:**

```
def mayor(a, b):
    if a > b:
        return a
    else:
        return b
```

Explicación Detallada:

1. **Comparación:** Se utiliza el operador de comparación `>` para determinar si el primer número (`a`) es mayor que el segundo (`b`).
2. **Devolución:** La estructura `if-else` devuelve el número mayor utilizando `return`. **Comandos Usados:**

- `>`: Operador de comparación.
- `if-else`: Condicional.
- `return`: Devuelve valor.

Ejercicio 5: Cálculo de Factorial Iterativo

Enunciado: Crea una función `factorial` que calcule el factorial de un número. **Solución:**

```
def factorial(n):
    resultado = 1
    for i in range(1, n + 1):
        resultado *= i
    return resultado
```

Explicación Detallada:

1. **Bucle:** Se utiliza el bucle `for i in range(1, n + 1)` para recorrer todos los números desde 1 hasta `n` (inclusive).
2. **Multiplicación Acumulativa:** El operador `*=` (multiplicación y asignación) multiplica el `resultado` actual por el número sucesivo (`i`) en cada iteración.

3. **Devolución:** Finalmente, `return` devuelve el factorial calculado. **Comandos Usados:**

- `for`: Define un bucle para iterar.
- `range()`: Genera una secuencia de números para la iteración.
- `*=` : Operador de multiplicación y asignación.

Bloque 2: Manipulación de Strings y Parámetros

Ejercicio 6: Saludo Personalizado

Enunciado: Crea una función `saludo_personalizado` que reciba un nombre y devuelva "Hola, [nombre]". **Solución:**

```
def saludo_personalizado(nombre):
    return "Hola, " + nombre
```

Explicación Detallada:

1. **Parámetro:** La función recibe el `nombre` como parámetro.
2. **Concatenación:** Se utiliza el operador `+` para unir (concatenar) el string fijo "Hola, " con el valor del parámetro `nombre`. **Comandos Usados:**

- `+`: Concatenar strings.
- `Parámetros`: Valores de entrada necesarios para la función.

Ejercicio 7: Potencia (Cuadrado)

Enunciado: Crea una función `cuadrado` que devuelva el cuadrado de un número. **Solución:**

```
def cuadrado(n):
    return n ** 2
```

Explicación Detallada:

1. **Potencia:** El operador `**` se utiliza para calcular la potencia de un número, elevando `n` a la potencia de 2.
2. **Retorno:** La función devuelve el valor calculado. **Comandos Usados:**

- `**`: Operador de potencia.
- `return`: Devuelve valor.

Ejercicio 8: Parámetro Opcional

Enunciado: Crea una función `saludo_opcional` que tenga un parámetro opcional `nombre` y devuelva "Hola, [nombre]". Si no se pasa nombre, que devuelva "Hola, invitado". **Solución:**

```
def saludo_opcional(nombre="invitado"):
    return "Hola, " + nombre
```

Explicación Detallada:

1. **Valor por Defecto:** Se define el parámetro con un valor asignado (`nombre="invitado"`). Si no se proporciona un argumento al llamar a la función, se utiliza este valor por defecto.
2. **Uso:** Si se pasa un argumento (ej: "Juan"), se usa ese valor; si no se pasa, se utiliza "invitado" para concatenar el string. **Comandos Usados:**

- `parametro=valor`: Define un valor por defecto para un parámetro.
- `return`: Devuelve valor.

Ejercicio 9: Suma de Elementos en una Lista

Enunciado: Crea una función `suma_lista` que reciba una lista de números y devuelva la suma de todos sus elementos. **Solución:**

```
def suma_lista(lista):
    total = 0
    for num in lista:
        total += num
    return total
```

Explicación Detallada:

1. **Recorrido:** La estructura `for num in lista` recorre cada uno de los elementos de la lista proporcionada.
2. **Acumulación:** El operador `+=` (suma y asigna) añade el valor de `num` al acumulador `total` en cada iteración.
3. **Resultado:** La función devuelve la suma final de todos los elementos. **Comandos Usados:**

- `for-in`: Recorre los elementos de una colección (lista).
- `+=`: Suma y asigna.
- `return`: Devuelve resultado.

Ejercicio 10: Longitud de una Cadena

Enunciado: Crea una función `longitud_cadena` que devuelva la longitud de un string. **Solución:**

```
def longitud_cadena(cadena):
    return len(cadena)
```

Explicación Detallada:

1. **Medición:** La función integrada `len()` se utiliza para obtener la longitud de la cadena de texto (string) o de una lista.
2. **Devolución:** El resultado se devuelve directamente usando `return`. **Comandos Usados:**

- `len()`: Mide la longitud de un string o lista.
- `return`: Devuelve valor.

Bloque 3: Lógica y Manejo Avanzado de Strings/Listas

Ejercicio 11: Verificación de Número Primo

Enunciado: Crea una función `es_primo` que reciba un número y devuelva `True` si es primo, `False` si no lo es. **Solución:**

```
def es_primo(n):
    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Explicación Detallada:

1. **Condición Inicial:** Se establece que los números menores que 2 no son primos.
2. **Búsqueda de Divisores:** Se recorre un rango desde 2 hasta el número-1 usando `for-in` y `range()`.
3. **Divisibilidad:** Si se encuentra algún divisor tal que el módulo (%) es 0, la función devuelve `False` y se detiene (no es primo).
4. **Primo:** Si el bucle termina sin encontrar divisores, se devuelve `True`. **Comandos Usados:**

- `for-in` y `range()`: Recorre números consecutivos.
- %: Verifica divisibilidad.
- `return`: Devuelve el resultado (`True` o `False`).

Ejercicio 12: Listas a Mayúsculas

Enunciado: Crea una función `mayusculas_lista` que reciba una lista de strings y devuelva otra lista con todas las palabras en mayúsculas. **Solución:**

```
def mayusculas_lista(lista):
    resultado = []
    for palabra in lista:
        resultado.append(palabra.upper())
    return resultado
```

Explicación Detallada:

1. **Conversión:** Dentro del bucle, `upper()` se aplica a cada `palabra` para convertir el string a mayúsculas.
2. **Construcción de Lista:** El comando `append()` se utiliza para agregar los strings convertidos a la nueva lista `resultado`.
3. **Devolución:** Se devuelve la nueva lista con todas las palabras en mayúsculas. **Comandos Usados:**

- `append()`: Agrega elementos a listas.
- `upper()`: Convierte un string a mayúsculas.

Ejercicio 13: Conteo de Vocales

Enunciado: Crea una función `contar_vocales` que reciba un string y devuelva cuántas vocales contiene. **Solución:**

```
def contar_vocales(cadena):
    contador = 0
    for letra in cadena.lower():
        if letra in "aeiou":
            contador += 1
    return contador
```

Explicación Detallada:

1. **Normalización:** `lower()` convierte todo el string a minúsculas, lo que simplifica la comparación.
2. **Pertenencia:** El comando `in` comprueba si una letra pertenece al conjunto de caracteres "aeiou".
3. **Conteo:** Si la letra es una vocal, el operador `+=` suma 1 al `contador`. **Comandos Usados:**

- `lower()`: Convierte string a minúsculas.
- `in`: Comprueba pertenencia a una colección.
- `+=`: Suma y asigna.

Ejercicio 14: Verificación de Palíndromo

Enunciado: Crea una función `es_palindromo` que verifique si un string es palíndromo (igual al revés). **Solución:**

```
def es_palindromo(cadena):
    cadena = cadena.replace(" ", "").lower()
    return cadena == cadena[::-1]
```

Explicación Detallada:

1. **Limpieza:** Primero, `replace(" ", "")` elimina cualquier espacio en blanco, y `lower()` asegura que la comparación no distinga entre mayúsculas y minúsculas.
2. **Inversión:** La técnica de *slicing* invertido `[::-1]` crea una versión de la cadena leída al revés.
3. **Comparación:** El operador `==` compara la cadena limpiada original con la cadena invertida. **Comandos Usados:**

- `replace()`: Reemplaza caracteres (aquí, elimina espacios).
- `[::-1]`: *Slice* invertido de string.
- `==`: Compara igualdad.

Ejercicio 15: Serie de Fibonacci

Enunciado: Crea una función `fibonacci` que reciba un número `n` y devuelva una lista con los primeros `n` números de Fibonacci. **Solución:**

```
def fibonacci(n):
    if n <= 0:
        return []
```

```
    elif n == 1:
        return
    fib =
    while len(fib) < n:
        fib.append(fib[-1] + fib[-2])
    return fib
```

Explicación Detallada:

1. **Casos Base:** Se manejan los casos de `n` menor o igual a 0, y `n` igual a 1.
2. **Bucle `while`:** El bucle `while` se repite mientras la longitud de la lista `fib` sea menor que `n`.
3. **Generación:** Se utiliza el *slicing* `fib[-1]` y `fib[-2]` para acceder a los dos últimos elementos. Su suma se agrega a la lista usando `append()`. **Comandos Usados:**

- `while`: Bucle que se repite mientras la condición sea verdadera.
- `append()`: Agrega un elemento a la lista.
- `Slicing [-1]`: Accede al último elemento de la lista.

Bloque 4: Operaciones con Listas y Bucles

Ejercicio 16: Producto de Elementos de una Lista

Enunciado: Crea una función `multiplicar_lista` que reciba una lista de números y devuelva su producto. **Solución:**

```
def multiplicar_lista(lista):
    resultado = 1
    for num in lista:
        resultado *= num
    return resultado
```

Explicación Detallada:

1. **Acumulación:** La variable `resultado` se inicializa a 1.
2. **Multiplicación:** El bucle `for-in` recorre la lista, y el operador `*=` multiplica cada número (`num`) al acumulador `resultado`.
3. **Devolución:** Devuelve el producto total. **Comandos Usados:**

- `*=` : Multiplicación y asignación.
- `for-in`: Recorre lista.

Ejercicio 17: Ordenar Lista

Enunciado: Crea una función `ordenar_lista` que reciba una lista de números y devuelva la lista ordenada de menor a mayor. **Solución:**

```
def ordenar_lista(lista):
    return sorted(lista)
```

Explicación Detallada:

1. **Función Incorporada:** Se utiliza la función `sorted()` que recibe la lista como argumento.

2. **Resultado:** `sorted()` devuelve una lista nueva que está ordenada, sin modificar la lista original. **Comandos Usados:**

- `sorted()`: Ordena listas o iterables.
- `return`: Devuelve el resultado.

Ejercicio 18: Tabla de Multiplicar

Enunciado: Crea una función `imprimir_tabla` que reciba un número y muestre su tabla de multiplicar del 1 al 10. **Solución:**

```
def imprimir_tabla(n):
    for i in range(1, 11):
        print(f"{n} x {i} = {n*i}")
```

Explicación Detallada:

1. **Rango:** `range(1, 11)` se utiliza para generar los números del 1 al 10 que servirán como multiplicadores.
2. **Formato:** Se usa un **f-string** (`f"{...}"`) para formatear la salida y mostrar la operación y el resultado de manera legible.
3. **Salida:** `print()` se usa para mostrar cada línea de la tabla. **Comandos Usados:**

- `for-in` y `range()`: Controlan el bucle.
- `f-string`: Permite insertar variables y expresiones directamente en strings.
- `print()`: Muestra datos.

Ejercicio 19: Mayoría de Edad

Enunciado: Crea una función `es_mayor_de_edad` que reciba la edad y devuelva `True` si es mayor de edad, `False` si no lo es. **Solución:**

```
def es_mayor_de_edad(edad):
    return edad >= 18
```

Explicación Detallada:

1. **Comparación:** El operador de comparación `>=` verifica si la `edad` es mayor o igual a 18.
2. **Devolución Directa:** La función devuelve directamente el resultado booleano de esta comparación. **Comandos Usados:**

- `>=`: Operador de comparación (mayor o igual).
- `return`: Devuelve valor.

Ejercicio 20: Conteo de Palabras

Enunciado: Crea una función `contar_palabras` que reciba un string y devuelva el número de palabras que contiene. **Solución:**

```
def contar_palabras(texto):
    palabras = texto.split()
    return len(palabras)
```

Explicación Detallada:

1. **División:** El método `split()` divide el string (`texto`) en una lista de palabras, utilizando los espacios como separadores por defecto.
2. **Conteo:** `len()` se utiliza para contar cuántos elementos (palabras) hay en la lista resultante. **Comandos Usados:**

- `split()`: Divide un string en una lista de palabras.
- `len()`: Devuelve el tamaño o longitud de la lista.

Bloque 5: Estadísticas y Funciones Incorporadas

Ejercicio 21: Encontrar el Máximo (Sin `max()`)

Enunciado: Crea una función `max_lista` que reciba una lista de números y devuelva el mayor valor, sin usar `max()`. **Solución:**

```
def max_lista(lista):
    mayor = lista
    for num in lista:
        if num > mayor:
            mayor = num
    return mayor
```

Explicación Detallada:

1. **Inicialización:** Se inicializa la variable `mayor` con el primer elemento de la lista.
2. **Iteración y Condición:** Se recorre la lista con `for-in`. Si se encuentra un número (`num`) que es mayor (`>`) que el valor actual de `mayor`, `mayor` se actualiza. **Comandos Usados:**

- `for-in`: Recorre lista.
- `if`: Condicional.
- `>`: Comparación.
- `return`: Devuelve el resultado.

Ejercicio 22: Cálculo de Promedio

Enunciado: Crea una función `promedio_lista` que reciba una lista de números y devuelva su promedio. **Solución:**

```
def promedio_lista(lista):
    total = sum(lista)
    return total / len(lista)
```

Explicación Detallada:

1. **Suma Total:** La función `sum()` suma todos los elementos de la lista.
2. **División:** El resultado se divide (`/`) por el número de elementos, obtenido mediante `len(lista)`, calculando así el promedio. **Comandos Usados:**

- `sum()`: Suma elementos de la lista.
- `len()`: Devuelve la longitud de la lista.
- `/`: Operador de división.
- `return`: Devuelve el resultado.

Ejercicio 23: Verificación de Año Bisiesto

Enunciado: Crea una función `es_bisiesto` que determine si un año es bisiesto. **Solución:**

```
def es_bisiesto(anio):
    return (anio % 4 == 0 and anio % 100 != 0) or (anio % 400 == 0)
```

Explicación Detallada:

1. **Lógica Bisiesta:** La condición se basa en la regla: un año es bisiesto si es divisible por 4 Y no divisible por 100, O si es divisible por 400.
2. **Operadores Lógicos:** Se utilizan los operadores `and` y `or` para combinar las condiciones de divisibilidad, junto con el operador módulo (%). **Comandos Usados:**

- %: Módulo.
- `and`, `or`: Operadores lógicos.
- `==`: Comparación de igualdad.
- `return`: Devuelve el valor booleano resultante.

Ejercicio 24: Invertir Lista

Enunciado: Crea una función `reversa_lista` que reciba una lista y devuelva otra con los elementos en orden inverso. **Solución:**

```
def reversa_lista(lista):
    return lista[::-1]
```

Explicación Detallada:

1. **Slicing Invertido:** Se utiliza el *slicing* especial `[::-1]`. Este comando genera una nueva lista conteniendo los elementos de la lista original en orden invertido. **Comandos Usados:**

- `Slicing [::-1]`: Invierte la lista o cadena.
- `return`: Devuelve el resultado.

Ejercicio 25: Intersección de Listas

Enunciado: Crea una función `interseccion_listas` que reciba dos listas y devuelva una lista con los elementos comunes. **Solución:**

```
def interseccion_listas(lista1, lista2):
    return [x for x in lista1 if x in lista2]
```

Explicación Detallada:

1. **List Comprehension:** Se utiliza la *list comprehension* para construir la nueva lista de forma concisa.
2. **Recorrido y Condición:** El comando `for x in lista1` recorre los elementos de la primera lista, y la condición `if x in lista2` verifica si ese elemento pertenece a la segunda lista (condición de pertenencia). **Comandos Usados:**

- `for x in lista1`: Recorre la lista.
- `if x in lista2`: Condición de pertenencia.
- `[]`: Define y crea la lista.

Bloque 6: Recursión y Conteo de Datos

Ejercicio 26: Factorial Recursivo

Enunciado: Crea una función recursiva `factorial_recursivo` que calcule el factorial de un número. **Solución:**

```
def factorial_recursivo(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial_recursivo(n - 1)
```

Explicación Detallada:

1. **Recursión:** Una función recursiva es aquella que se llama a sí misma.
2. **Caso Base:** La condición `if n == 0 or n == 1` establece el caso base, que es el punto donde la recursión se detiene (devuelve 1).
3. **Llamada Recursiva:** En el `else`, la función se llama a sí misma con `n - 1`, multiplicando `n` por el resultado de la llamada anterior. **Comandos Usados:**

- `Recursión`: Función que se llama a sí misma.
- `if-else`: Condicional.
- `*`: Multiplicación.
- `return`: Devuelve el resultado.

Ejercicio 27: Contar Elementos Específicos

Enunciado: Crea una función `contar_elementos` que reciba una lista y un valor, y cuente cuántas veces aparece ese valor. **Solución:**

```
def contar_elementos(lista, valor):
    contador = 0
    for elem in lista:
        if elem == valor:
            contador += 1
    return contador
```

Explicación Detallada:

1. **Recorrido:** El bucle `for-in` recorre cada elemento (`elem`) en la lista.
2. **Comparación:** El operador `==` compara si el elemento actual coincide con el `valor` buscado.
3. **Conteo:** Si hay coincidencia, el operador `+=` suma 1 al `contador`. **Comandos Usados:**

- `for-in`: Recorre lista.
- `==`: Compara valores.
- `+=`: Suma y asigna.
- `return`: Devuelve el resultado.

Ejercicio 28: Eliminar Duplicados

Enunciado: Crea una función `eliminar_repetidos` que reciba una lista y devuelva otra sin elementos duplicados. **Solución:**

```
def eliminar_repetidos(lista):
    return list(set(lista))
```

Explicación Detallada:

1. **Conjunto Único:** La función `set()` convierte la lista en un *conjunto* de elementos, lo cual inherentemente elimina todos los duplicados.
2. **Reconversión:** `list()` se utiliza para convertir el conjunto de vuelta a una lista. **Comandos Usados:**

- `set()`: Crea un conjunto, eliminando elementos duplicados.
- `list()`: Convierte el conjunto de nuevo a lista.
- `return`: Devuelve el resultado.

Ejercicio 29: Palabras Más Largas

Enunciado: Crea una función `palabras_mas_largas` que reciba una lista de palabras y devuelva las de mayor longitud. **Solución:**

```
def palabras_mas_largas(lista):
    max_len = max(len(p) for p in lista)
    return [p for p in lista if len(p) == max_len]
```

Explicación Detallada:

1. **Cálculo de Máximo:** Se calcula la longitud máxima (`max_len`) utilizando `max()` y la función `len()`.
2. **Filtrado:** Una segunda *List comprehension* filtra la lista original, seleccionando solo aquellas palabras cuya longitud es exactamente igual a `max_len`. **Comandos Usados:**

- `len()`: Longitud de string.
- `max()`: Devuelve el valor máximo.
- `List comprehension`: Crea una lista basada en una condición y una iteración.

Ejercicio 30: Suma de Listas por Posición

Enunciado: Crea una función `suma_dos_listas` que reciba dos listas del mismo tamaño y devuelva una lista con la suma de sus elementos por posición. **Solución:**

```
def suma_dos_listas(lista1, lista2):
    return [a + b for a, b in zip(lista1, lista2)]
```

Explicación Detallada:

1. **Combinación:** La función `zip()` combina los elementos de ambas listas en pares, según su posición (ej: `(lista1, lista2)`).

2. **List Comprehension y Suma:** Se utiliza *list comprehension* junto con `for a,b in zip(...)` para iterar sobre estos pares y sumar `a + b` para crear la nueva lista. **Comandos Usados:**

- `zip()`: Combina listas o iterables por posición.
- `for a,b in zip()`: Recorre los elementos emparejados.
- `[]`: Crea lista.
- `return`: Devuelve resultado.

Bloque 7: Parámetros Flexibles y Tuplas

Ejercicio 31: Suma de N Argumentos

Enunciado: Crea una función `sumar_n` que reciba cualquier cantidad de números y devuelva su suma. **Solución:**

```
def sumar_n(*numeros):
    return sum(numeros)
```

Explicación Detallada:

1. **Argumentos Múltiples:** El parámetro `*numeros` (conocido como `*args`) permite que la función acepte un número arbitrario de argumentos posicionales, los cuales se empaquetan en una tupla llamada `numeros`.
2. **Suma Directa:** La función `sum()` suma directamente todos los elementos de la tupla `numeros`. **Comandos Usados:**

- `*args` (o `*numeros`): Parámetro que acepta múltiples argumentos.
- `sum()`: Suma elementos.
- `return`: Devuelve resultado.

Ejercicio 32: Formateo de Información

Enunciado: Crea una función `info_persona` que reciba nombre y edad, y devuelva un string con la información formateada. **Solución:**

```
def info_persona(nombre, edad):
    return f"Nombre: {nombre}, Edad: {edad}"
```

Explicación Detallada:

1. **f-string:** Se utiliza un **f-string** (`f"{...}"`) que permite la inserción simple y directa de variables (`nombre` y `edad`) dentro del string resultante. **Comandos Usados:**

- `f"{variable}"`: Sintaxis para formatear string (f-string).
- `return`: Devuelve el string formateado.

Ejercicio 33: Mínimo y Máximo en Tupla

Enunciado: Crea una función `min_max_lista` que reciba una lista de números y devuelva una tupla con el mínimo y máximo. **Solución:**

```
def min_max_lista(lista):
    return (min(lista), max(lista))
```

Explicación Detallada:

1. **Funciones de Agregación:** Se usan las funciones incorporadas `min()` y `max()` para obtener los valores mínimo y máximo, respectivamente.
2. **Devolución en Tupla:** Los resultados se agrupan y se devuelven como una tupla, definida por los paréntesis `()`. **Comandos Usados:**

- `min()`, `max()`: Devuelven el mínimo y el máximo.
- `()`: Define una tupla.
- `return`: Devuelve el resultado.

Ejercicio 34: Filtrar Números Pares

Enunciado: Crea una función `filtrar_pares` que reciba una lista y devuelva otra solo con los números pares. **Solución:**

```
def filtrar_pares(lista):
    return [x for x in lista if x % 2 == 0]
```

Explicación Detallada:

1. **List Comprehension con Filtro:** Se utiliza *list comprehension* para iterar sobre `lista`.
2. **Condición de Paridad:** Solo se incluyen los elementos `x` que cumplen la condición de que su módulo con 2 sea 0 (`x % 2 == 0`). **Comandos Usados:**

- `%`: Operador módulo.
- `==`: Comparación.
- `List comprehension`: Crea una lista según una condición.
- `return`: Devuelve el resultado.

Bloque 8: Diccionarios y Expresiones Lambda

Ejercicio 35: Ordenar Diccionario por Valor

Enunciado: Crea una función `ordenar_diccionario` que reciba un diccionario y devuelva una lista de tuplas ordenadas por valor. **Solución:**

```
def ordenar_diccionario(dic):
    return sorted(dic.items(), key=lambda x: x)
```

Explicación Detallada:

1. **Pares Clave-Valor:** `dic.items()` transforma el diccionario en una lista de tuplas `(clave, valor)`.
2. **Función `lambda`:** Se usa una `lambda` (función anónima) dentro de `sorted()` para especificar que la clave de ordenación (`key`) debe ser el segundo elemento de la tupla (`x`), que corresponde al valor del diccionario.

3. **Ordenación:** `sorted()` ordena la lista de tuplas basándose en el valor. **Comandos Usados:**

- `dict.items()`: Devuelve tuplas clave-valor.
- `sorted()`: Ordena la lista.
- `lambda`: Define una función anónima.
- `return`: Devuelve el resultado.

Ejercicio 36: Verificación de Capicúa (Número)

Enunciado: Crea una función `es_capicua_num` que reciba un número y devuelva `True` si es capicúa. **Solución:**

```
def es_capicua_num(n):
    n_str = str(n)
    return n_str == n_str[::-1]
```

Explicación Detallada:

1. **Conversión:** Se utiliza `str()` para convertir el número a string, ya que los strings son fáciles de invertir.
2. **Inversión y Comparación:** Se compara la cadena original con su versión invertida (`[::-1]`). **Comandos Usados:**

- `str()`: Convierte a string.
- `[::-1]`: Invierte cadena.
- `==`: Comparación.
- `return`: Devuelve resultado.

Ejercicio 37: Invertir Claves y Valores de Diccionario

Enunciado: Crea una función `diccionario_invertido` que reciba un diccionario y devuelva otro con claves y valores intercambiados. **Solución:**

```
def diccionario_invertido(dic):
    return {v: k for k, v in dic.items()}
```

Explicación Detallada:

1. **Dictionary Comprehension:** Se utiliza la *dictionary comprehension* para crear el nuevo diccionario.
2. **Intercambio:** Al iterar sobre `dic.items()`, se establece el formato `{v: k}`, donde `v` (el valor original) se convierte en la nueva clave, y `k` (la clave original) se convierte en el nuevo valor. **Comandos Usados:**

- `{v:k for k,v in dic.items()}`: Sintaxis de diccionario por comprensión.
- `dict.items()`: Tuplas clave-valor.
- `return`: Devuelve resultado.

Ejercicio 38: Aplicar Función a Lista

Enunciado: Crea una función `aplicar_funcion_lista` que reciba una lista y una función, y devuelva la lista con la función aplicada a cada elemento. **Solución:**

```
def aplicar_funcion_lista(lista, funcion):
    return [funcion(x) for x in lista]
```

Explicación Detallada:

1. **Función como Parámetro:** La función `aplicar_funcion_lista` acepta otra función (`funcion`) como uno de sus parámetros.
2. **List Comprehension:** Se usa *list comprehension* para iterar sobre la lista, aplicando la `funcion(x)` a cada elemento `x` y construyendo la nueva lista de resultados. **Comandos Usados:**

- `lambda`: Se puede usar para pasar funciones anónimas.
- `List comprehension`: Para aplicar una operación a cada elemento de la lista.
- `return`: Devuelve resultado.

Ejercicio 39: Sumar Valores de Diccionario

Enunciado: Crea una función `sumar_diccionario` que reciba un diccionario y devuelva la suma de sus valores. **Solución:**

```
def sumar_diccionario(dic):
    return sum(dic.values())
```

Explicación Detallada:

1. **Obtener Valores:** `dic.values()` devuelve una vista de todos los valores contenidos en el diccionario.
2. **Suma:** La función `sum()` calcula la suma total de esta colección de valores. **Comandos Usados:**

- `dict.values()`: Obtiene los valores del diccionario.
- `sum()`: Suma elementos.
- `return`: Devuelve resultado.

Bloque 9: Recursión y Funciones Anidadas

Ejercicio 40: Suma Recursiva (1 a N)

Enunciado: Crea una función `recursiva_suma_n` que reciba un número `n` y devuelva la suma de todos los números desde 1 hasta `n` usando recursión. **Solución:**

```
def recursiva_suma_n(n):
    if n <= 0:
        return 0
    else:
        return n + recursiva_suma_n(n - 1)
```

Explicación Detallada:

1. **Caso Base:** Si `n` es menor o igual a 0, la suma se detiene y devuelve 0.
2. **Llamada Recursiva:** La función se llama a sí misma con `n - 1`. El valor de `n` actual se suma (`+`) al resultado de la función anterior, sumando todos los valores en el retorno de la recursión. **Comandos Usados:**

- `Recursión`: Función que se llama a sí misma.
- `if-else`: Condicional.
- `+`: Suma.
- `return`: Devuelve resultado.

Ejercicio 41: Función Anidada

Enunciado: Crea una función `funcion_anidada` que contenga otra función dentro y que la función interna sume 10 a un número. **Solución:**

```
def funcion_anidada(n):
    def sumar_diez(x):
        return x + 10
    return sumar_diez(n)
```

Explicación Detallada:

1. **Definición Interna:** Se utiliza `def` para definir la función `sumar_diez(x)` dentro de `funcion_anidada(n)`. Esto crea una *Función anidada*.
2. **Ejecución:** La función principal llama y ejecuta la función interna `sumar_diez(n)`. **Comandos Usados:**

- `def`: Define función.
- `Función anidada`: Función definida dentro de otra.
- `return`: Devuelve resultado.

Ejercicio 42: Filtrar Diccionario por Valor Mínimo

Enunciado: Crea una función `filtrar_diccionario` que reciba un diccionario y un valor mínimo, y devuelva un diccionario con valores mayores o iguales al mínimo. **Solución:**

```
def filtrar_diccionario(dic, min_valor):
    return {k:v for k,v in dic.items() if v >= min_valor}
```

Explicación Detallada:

1. **Dictionary Comprehension con Condición:** Se utiliza la comprensión de diccionario.
2. **Filtro:** Se recorren los ítems (`k,v`) y se incluye un par clave-valor solo si el valor (`v`) es mayor o igual (`>=`) al `min_valor`. **Comandos Usados:**

- `{k:v for k,v in dic.items()}`: Sintaxis de diccionario por comprensión.
- `>=`: Comparación de mayor o igual.
- `return`: Devuelve resultado.

Ejercicio 43: Aplicar Múltiples Funciones

Enunciado: Crea una función `aplicar_varias_funciones` que reciba un número y una lista de funciones, y devuelva una lista con los resultados. **Solución:**

```
def aplicar_varias_funciones(n, funciones):
    return [f(n) for f in funciones]
```

Explicación Detallada:

1. **Lista de Funciones:** La función recibe una lista que contiene objetos de tipo función (como las definidas con `lambda`).
2. **List Comprehension:** Se itera sobre la lista de `funciones`. En cada paso, la función actual (`f`) se aplica al número `n` (`f(n)`), y el resultado se agrega a la lista de retorno.
   Comandos Usados:

- `List comprehension`: Crea la lista de resultados aplicando la lógica iterativa.
- `lambda`: Función anónima (usada en el ejemplo de solución).
- `return`: Devuelve resultado.

Ejercicio 44: Conteo de Ocurrencias (Frecuencia)

Enunciado: Crea una función `contar_ocurrencias` que reciba una lista de strings y devuelva un diccionario con la cantidad de veces que aparece cada palabra. **Solución:**

```
def contar_ocurrencias(lista):
    resultado = {}
    for palabra in lista:
        if palabra in resultado:
            resultado[palabra] += 1
        else:
            resultado[palabra] = 1
    return resultado
```

Explicación Detallada:

1. **Estructura:** Se utiliza un `Diccionario` para almacenar la cuenta (clave=palabra, valor=cantidad).
2. **Verificación de Clave:** El comando `in` verifica si la `palabra` ya existe como clave en el diccionario.
3. **Acumulación:** Si existe, `+=` incrementa su valor en 1; si no existe, se inicializa a 1.
   Comandos Usados:

- `for-in`: Recorre lista.
- `in`: Verifica existencia de clave.
- `+=`: Suma y asigna.
- `Diccionario`: Estructura clave-valor.

Bloque 10: Decoradores, Estructuras Complejas y Transformaciones

Ejercicio 45: Decorador Simple

Enunciado: Crea una función `decorador_simple` que reciba otra función y la ejecute mostrando un mensaje antes y después. **Solución:**

```
def decorador_simple(func):
    def nueva_funcion():
        print("Antes de la función")
        func()
        print("Después de la función")
    return nueva_funcion
```

Explicación Detallada:

1. **Decorador:** Un decorador es una función que recibe otra `func` como parámetro y la modifica.
2. **Función Anidada:** Se define `nueva_funcion()` dentro del decorador. Esta función interna contiene la lógica de envoltura (`print` antes, ejecución de `func()`, y `print` después).
3. **Retorno de Función:** El decorador devuelve la `nueva_funcion` modificada, no su resultado. **Comandos Usados:**

- `Función como parámetro`.
- `Función anidada`.
- `print()`: Muestra texto.
- `return`: Devuelve una función.

Ejercicio 46: Aplanar Lista de Listas

Enunciado: Crea una función `aplanar_lista` que reciba una lista de listas y devuelva una lista plana con todos los elementos. **Solución:**

```
def aplanar_lista(lista):
    return [elem for sublista in lista for elem in sublista]
```

Explicación Detallada:

1. **List Comprehension Doble Anidada:** Se utiliza una sintaxis de *List comprehension* doble.
2. **Iteración:** `for sublista in lista` itera sobre las listas internas, y `for elem in sublista` extrae cada elemento individualmente, agregándolo directamente a la lista plana resultante. **Comandos Usados:**

- `List comprehension doble`: Para iterar sobre estructuras anidadas.
- `return`: Devuelve resultado.

Ejercicio 47: Diccionario de Factoriales Recursivo

Enunciado: Crea una función `funcion_recursiva_diccionario` que reciba un número `n` y devuelva un diccionario con `clave=numero` y `valor=factorial` usando recursión.
Solución:

```
def funcion_recursiva_diccionario(n):
    if n == 0:
        return {0:1}
    else:
        dic = funcion_recursiva_diccionario(n-1)
        dic[n] = n * dic[n-1]
        return dic
```

Explicación Detallada:

1. **Recursión y Diccionario:** La función utiliza la `Recursión` para construir el diccionario incrementalmente.
2. **Caso Base:** Si `n` es 0, devuelve el caso inicial `{0: 1}`.
3. **Construcción:** En la llamada recursiva, se recupera el diccionario generado para `n-1` (`dic`). Luego, se calcula el factorial de `n` multiplicando `n` por el factorial de `n-1` (almacenado en `dic[n-1]`), y se asigna esta nueva entrada al diccionario. **Comandos Usados:**

- `Recursión`.
- `Diccionario`.
- `*`: Multiplicación.
- `return`: Devuelve resultado.

Ejercicio 48: Filtrar Palabras por Longitud

Enunciado: Crea una función `filtrar_palabras` que reciba un string y devuelva una lista con palabras mayores a 3 letras. **Solución:**

```
def filtrar_palabras(texto):
    return [palabra for palabra in texto.split() if len(palabra) > 3]
```

Explicación Detallada:

1. **División:** `texto.split()` separa el string en palabras individuales.
2. **List Comprehension:** Se utiliza esta técnica para iterar sobre las palabras.
3. **Filtro:** Se incluye una palabra solo si su longitud (`len()`) es estrictamente mayor a 3. **Comandos Usados:**

- `split()`: Separa palabras.
- `len()`: Longitud.
- `List comprehension`: Crea la lista filtrada.
- `return`: Devuelve resultado.

Ejercicio 49: Transformar Valores de Diccionario

Enunciado: Crea una función `transformar_diccionario` que reciba un diccionario y devuelva otro multiplicando sus valores por 2. **Solución:**

```
def transformar_diccionario(dic):
    return {k:v*2 for k,v in dic.items()}
```

Explicación Detallada:

1. **Dictionary Comprehension:** Permite crear un nuevo diccionario de manera eficiente.
2. **Transformación:** Se recorren los pares clave (`k`) y valor (`v`). La clave se mantiene, pero el nuevo valor se calcula como `v*2`. **Comandos Usados:**

- `{k:v for k,v in dic.items()}`: Diccionario por comprensión.
- `*`: Multiplicación.
- `return`: Devuelve resultado.

Ejercicio 50: Aplicar Funciones Anidadas en Secuencia

Enunciado: Crea una función `aplicar_funciones_anidadas` que reciba un número y aplique varias funciones internas en secuencia, devolviendo el resultado final. **Solución:**

```
def aplicar_funciones_anidadas(n):
    def sumar_dos(x):
        return x + 2
    def multiplicar_tres(x):
        return x * 3
    return multiplicar_tres(sumar_dos(n))
```

Explicación Detallada:

1. **Funciones Anidadas:** Se definen dos funciones internas (`sumar_dos` y `multiplicar_tres`).
2. **Secuencia de Ejecución:** La función principal ejecuta las operaciones en secuencia (primero `sumar_dos(n)`, y luego ese resultado se pasa a `multiplicar_tres`).
3. **Operadores:** Se utilizan los operadores aritméticos `+` (suma) y `*` (multiplicación). **Comandos Usados:**

- `Funciones anidadas`.
- `return`: Devuelve resultado.
- `*`: Multiplicación.
- `+`: Suma.

Ejercicio 1

Enunciado:
Crea una función `saludo` que imprima "Hola, mundo".

Solución:

```
def saludo():
    print("Hola, mundo")

saludo()
```

Explicación:

- Se define una función usando `def` seguida del nombre de la función y paréntesis.
- `print()` se usa para mostrar texto en pantalla.
- Llamamos a la función con `saludo()` para ejecutar el código dentro de la función.

Comandos usados:

- `def`: Define una función.
- `print()`: Muestra texto o datos en pantalla.

Ejercicio 2

Enunciado:
Crea una función `suma` que reciba dos números y devuelva su suma.

Solución:

```
def suma(a, b):
    return a + b

resultado = suma(3, 5)
print(resultado)
```

Explicación:

- `a` y `b` son **parámetros** de la función.
- `return` devuelve el resultado de la suma.
- Llamamos a la función con `suma(3, 5)` y almacenamos el resultado en una variable.

Comandos usados:

- `return`: Devuelve un valor de la función.
- Variables: almacenan datos.
- `print()`: muestra resultados.

Ejercicio 3

Enunciado:
Crea una función `es_par` que reciba un número y devuelva `True` si es par, `False` si es impar.

Solución:

```
def es_par(num):
    if num % 2 == 0:
        return True
    else:
        return False

print(es_par(4))
print(es_par(7))
```

Explicación:

- `%` calcula el **resto** de la división.
- `if` verifica una condición.
- La función devuelve `True` o `False` según sea par o impar.

Comandos usados:

- `%`: operador módulo.
- `if-else`: condicional.
- `return`: devuelve un valor.

Ejercicio 4

Enunciado:
Crea una función `mayor` que reciba dos números y devuelva el mayor.

Solución:

```
def mayor(a, b):
    if a > b:
        return a
    else:
        return b

print(mayor(10, 7))
```

Explicación:

- Compara dos números usando `>`.
- Devuelve el número mayor usando `return`.

Comandos usados:

- `>`: operador de comparación.
- `if-else`: condicional.
- `return`: devuelve valor.

Ejercicio 5

Enunciado:
Crea una función `factorial` que calcule el factorial de un número.

Solución:

```
def factorial(n):
    resultado = 1
    for i in range(1, n + 1):
        resultado *= i
    return resultado

print(factorial(5))
```

Explicación:

- `for i in range(1, n+1)`: recorre del 1 al n.
- `resultado *= i` multiplica cada número sucesivo.
- `return` devuelve el factorial calculado.

Comandos usados:

- `for`: bucle.
- `range()`: genera una secuencia de números.
- `*=`: operador de multiplicación y asignación.

Ejercicio 6

Enunciado:
Crea una función `saludo_personalizado` que reciba un nombre y devuelva "Hola, [nombre]".

Solución:

```
def saludo_personalizado(nombre):
    return "Hola, " + nombre

print(saludo_personalizado("Ana"))
```

Explicación:

- Concatenamos texto con `+`.
- `nombre` es el parámetro de la función.
- Devuelve el saludo personalizado.

Comandos usados:

- `+`: concatenar strings.
- Parámetros: valores de entrada a la función.

Ejercicio 7

Enunciado:
Crea una función `cuadrado` que devuelva el cuadrado de un número.

Solución:

```
def cuadrado(n):
    return n ** 2

print(cuadrado(4))
```

Explicación:

- `**` es el operador de potencia.
- La función recibe un número y devuelve su cuadrado.

Comandos usados:

- `**`: potencia.
- `return`: devuelve valor.

Ejercicio 8

Enunciado:
Crea una función `saludo_opcional` que tenga un parámetro opcional `nombre` y devuelva "Hola, [nombre]". Si no se pasa nombre, que devuelva "Hola, invitado".

Solución:

```
def saludo_opcional(nombre="invitado"):
    return "Hola, " + nombre

print(saludo_opcional("Juan"))
print(saludo_opcional())
```

Explicación:

- Los parámetros pueden tener **valor por defecto**.
- Si no se pasa argumento, se usa el valor por defecto.

Comandos usados:

- `parametro=valor`: define valor por defecto.
- `return`: devuelve valor.

Ejercicio 9

Enunciado:
Crea una función `suma_lista` que reciba una lista de números y devuelva la suma de todos sus elementos.

Solución:

```
def suma_lista(lista):
    total = 0
    for num in lista:
        total += num
    return total

print(suma_lista([1, 2, 3, 4]))
```

Explicación:

- `for num in lista` recorre todos los elementos de la lista.
- `+=` suma y asigna a la variable `total`.
- Devuelve la suma de todos los elementos.

Comandos usados:

- `for-in`: recorre elementos de una colección.
- `+=`: suma y asigna.
- `return`: devuelve resultado.

Ejercicio 10

Enunciado:
Crea una función `longitud_cadena` que devuelva la longitud de un string.

Solución:

```
def longitud_cadena(cadena):
    return len(cadena)

print(longitud_cadena("Python"))
```

Explicación:

- `len()` devuelve la longitud de un string o lista.
- La función devuelve la longitud usando `return`.

Comandos usados:

- `len()`: mide longitud de string o lista.
- `return`: devuelve valor.

Ejercicio 11

Enunciado:
Crea una función `es_primo` que reciba un número y devuelva `True` si es primo, `False` si no lo es.

Solución:

```
def es_primo(n):
    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True


print(es_primo(7))
print(es_primo(10))
```

Explicación:

- Los números menores que 2 no son primos.
- Se recorre del 2 al número-1; si hay un divisor, no es primo.
- `return` devuelve `True` o `False`.

Comandos usados:

- `for-in` y `range()`: recorre números consecutivos.
- `%`: verifica divisibilidad.
- `return`: devuelve el resultado.

Ejercicio 12

Enunciado:
Crea una función `mayusculas_lista` que reciba una lista de strings y devuelva otra lista con todas las palabras en mayúsculas.

Solución:

```
def mayusculas_lista(lista):
    resultado = []
    for palabra in lista:
        resultado.append(palabra.upper())
    return resultado


print(mayusculas_lista(["hola", "mundo"]))
```

Explicación:

- `upper()` convierte un string a mayúsculas.
- `append()` agrega elementos a la lista.
- Se devuelve la nueva lista.

Comandos usados:

- `append()`: agrega elementos a listas.
- `upper()`: convierte string a mayúsculas.

Ejercicio 13

Enunciado:
Crea una función `contar_vocales` que reciba un string y devuelva cuántas vocales contiene.

Solución:

```
def contar_vocales(cadena):
    contador = 0
    for letra in cadena.lower():
        if letra in "aeiou":
            contador += 1
    return contador


print(contar_vocales("Python es divertido"))
```

Explicación:

- `lower()` convierte todo a minúsculas para facilitar la comparación.
- `in` verifica si un elemento pertenece a un conjunto de caracteres.
- Se suma a `contador` cada vocal encontrada.

Comandos usados:

- `lower()`: convierte string a minúsculas.
- `in`: comprueba pertenencia.
- `+=`: suma y asigna.

Ejercicio 14

Enunciado:
Crea una función `es_palindromo` que verifique si un string es palíndromo (igual al revés).

Solución:

```
def es_palindromo(cadena):
    cadena = cadena.replace(" ", "").lower()
    return cadena == cadena[::-1]


print(es_palindromo("Ana"))
print(es_palindromo("Hola"))
```

Explicación:

- `replace(" ", "")` elimina espacios.
- `[::-1]` invierte la cadena.
- Se compara la cadena original con la invertida.

Comandos usados:

- `replace()`: reemplaza caracteres.
- `[::-1]`: slice invertido de string.
- `==`: compara igualdad.

Ejercicio 15

Enunciado:
Crea una función `fibonacci` que reciba un número `n` y devuelva una lista con los primeros `n` números de Fibonacci.

Solución:

```
def fibonacci(n):
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    fib = [0, 1]
    while len(fib) < n:
        fib.append(fib[-1] + fib[-2])
    return fib


print(fibonacci(7))
```

Explicación:

- `fib[-1]` y `fib[-2]` acceden a los dos últimos elementos de la lista.
- Se usa `while` para generar hasta `n` elementos.

Comandos usados:

- `while`: bucle que se repite mientras la condición sea verdadera.
- `append()`: agrega a lista.
- Slicing `[-1]`: último elemento.

Ejercicio 16

Enunciado:
Crea una función `multiplicar_lista` que reciba una lista de números y devuelva su producto.

Solución:

```
def multiplicar_lista(lista):
    resultado = 1
    for num in lista:
        resultado *= num
    return resultado


print(multiplicar_lista([1, 2, 3, 4]))
```

Explicación:

- Se multiplica cada número de la lista al acumulador `resultado`.
- Devuelve el producto total.

Comandos usados:

- `*=`: multiplicación y asignación.
- `for-in`: recorre lista.

Ejercicio 17

Enunciado:
Crea una función `ordenar_lista` que reciba una lista de números y devuelva la lista ordenada de menor a mayor.

Solución:

```
def ordenar_lista(lista):
    return sorted(lista)


print(ordenar_lista([5, 2, 9, 1]))
```

Explicación:

- `sorted()` devuelve una lista ordenada sin modificar la original.

Comandos usados:

- `sorted()`: ordena listas o iterables.
- `return`: devuelve resultado.

Ejercicio 18

Enunciado:
Crea una función `imprimir_tabla` que reciba un número y muestre su tabla de multiplicar del 1 al 10.

Solución:

```
def imprimir_tabla(n):
    for i in range(1, 11):
        print(f"{n} x {i} = {n*i}")


imprimir_tabla(5)
```

Explicación:

- `f"{n} x {i} = {n*i}"` es un **f-string** para formatear texto.
- `range(1, 11)` recorre del 1 al 10.

Comandos usados:

- `for-in` y `range()`: bucle.
- `f-string`: inserta variables en strings.
- `print()`: mostrar datos.

Ejercicio 19

Enunciado:
Crea una función `es_mayor_de_edad` que reciba la edad y devuelva `True` si es mayor de edad, `False` si no lo es.

Solución:

```
def es_mayor_de_edad(edad):
    return edad >= 18


print(es_mayor_de_edad(20))
print(es_mayor_de_edad(16))
```

Explicación:

- `>=` compara si un número es mayor o igual.
- La función devuelve directamente el resultado de la comparación.

Comandos usados:

- `>=`: operador de comparación.
- `return`: devuelve valor.

Ejercicio 20

Enunciado:
Crea una función `contar_palabras` que reciba un string y devuelva el número de palabras que contiene.

Solución:

```
def contar_palabras(texto):
    palabras = texto.split()
    return len(palabras)

print(contar_palabras("Python es divertido"))
```

Explicación:

- `split()` divide un string en palabras usando espacios por defecto.
- `len()` cuenta cuántas palabras hay.

Comandos usados:

- `split()`: divide string en lista de palabras.
- `len()`: devuelve tamaño de lista o string.

Ejercicio 21

Enunciado:
Crea una función `max_lista` que reciba una lista de números y devuelva el mayor valor, sin usar `max()`.

Solución:

```
def max_lista(lista):
    mayor = lista[0]
    for num in lista:
        if num > mayor:
            mayor = num
    return mayor

print(max_lista([3, 7, 2, 9, 5]))
```

Explicación:

- Inicializamos `mayor` con el primer elemento.
- Recorremos la lista y actualizamos `mayor` si encontramos un número mayor.
- `return` devuelve el valor máximo.

Comandos usados:

- `for-in`: recorre lista.
- `if`: condicional.
- `>`: comparación.
- `return`: devuelve resultado.

Ejercicio 22

Enunciado:
Crea una función `promedio_lista` que reciba una lista de números y devuelva su promedio.

Solución:

```
def promedio_lista(lista):
    total = sum(lista)
    return total / len(lista)

print(promedio_lista([5, 10, 15]))
```

Explicación:

- `sum(lista)` suma todos los elementos.
- `len(lista)` devuelve el número de elementos.
- La división calcula el promedio.

Comandos usados:

- `sum()`: suma elementos de la lista.
- `len()`: longitud de lista.
- `/`: división.
- `return`: devuelve resultado.

Ejercicio 23

Enunciado:
Crea una función `es_bisiesto` que determine si un año es bisiesto.

Solución:

```
def es_bisiesto(anio):
    return (anio % 4 == 0 and anio % 100 != 0) or (anio % 400 == 0)


print(es_bisiesto(2024))
print(es_bisiesto(1900))
```

Explicación:

- Los años bisiestos son divisibles por 4 y no por 100, salvo si son divisibles por 400.
- La función devuelve directamente el resultado de la condición.

Comandos usados:

- `%`: módulo.
- `and`, `or`: operadores lógicos.
- `==`: comparación.
- `return`: devuelve valor.

Ejercicio 24

Enunciado:
Crea una función `reversa_lista` que reciba una lista y devuelva otra con los elementos en orden inverso.

Solución:

```
def reversa_lista(lista):
    return lista[::-1]


print(reversa_lista([1, 2, 3, 4]))
```

Explicación:

- `[::-1]` devuelve la lista invertida usando slicing.

Comandos usados:

- Slicing `[::-1]`: invierte lista.
- `return`: devuelve resultado.

Ejercicio 25

Enunciado:
Crea una función `interseccion_listas` que reciba dos listas y devuelva una lista con los elementos comunes.

Solución:

```
def interseccion_listas(lista1, lista2):
    return [x for x in lista1 if x in lista2]


print(interseccion_listas([1,2,3],[2,3,4]))
```

Explicación:

- Se usa **list comprehension** para crear una lista con elementos de `lista1` que estén en `lista2`.

Comandos usados:

- `for x in lista1`: recorre lista.
- `if x in lista2`: condición de pertenencia.
- `[]`: define lista.

Ejercicio 26

Enunciado:
Crea una función recursiva `factorial_recursivo` que calcule el factorial de un número.

Solución:

```
def factorial_recursivo(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial_recursivo(n - 1)


print(factorial_recursivo(5))
```

Explicación:

- Una función **recursiva** se llama a sí misma.
- Caso base: `n == 0` o `n == 1`.
- Se multiplica `n` por el factorial del número anterior.

Comandos usados:

- Recursión: función que se llama a sí misma.
- `if-else`: condicional.
- `*`: multiplicación.
- `return`: devuelve resultado.

Ejercicio 27

Enunciado:
Crea una función `contar_elementos` que reciba una lista y un valor, y cuente cuántas veces aparece ese valor.

Solución:

```
def contar_elementos(lista, valor):
    contador = 0
    for elem in lista:
        if elem == valor:
            contador += 1
    return contador


print(contar_elementos([1,2,2,3,2],2))
```

Explicación:

- Recorre la lista y suma a `contador` cada coincidencia con `valor`.

Comandos usados:

- `for-in`: recorre lista.
- `==`: compara valores.
- `+=`: suma y asigna.
- `return`: devuelve resultado.

Ejercicio 28

Enunciado:
Crea una función `eliminar_repetidos` que reciba una lista y devuelva otra sin elementos duplicados.

Solución:

```
def eliminar_repetidos(lista):
    return list(set(lista))


print(eliminar_repetidos([1,2,2,3,3,3,4]))
```

Explicación:

- `set()` elimina duplicados, `list()` convierte de nuevo a lista.

Comandos usados:

- `set()`: conjunto de elementos únicos.
- `list()`: convierte a lista.
- `return`: devuelve resultado.

Ejercicio 29

Enunciado:
Crea una función `palabras_mas_largas` que reciba una lista de palabras y devuelva las de mayor longitud.

Solución:

```
def palabras_mas_largas(lista):
    max_len = max(len(p) for p in lista)
    return [p for p in lista if len(p) == max_len]


print(palabras_mas_largas(["hola","adiós","mundo","Python"]))
```

Explicación:

- Se calcula la longitud máxima con `max()`.
- Se seleccionan las palabras que tengan esa longitud.

Comandos usados:

- `len()`: longitud de string.
- `max()`: devuelve máximo.
- List comprehension: crea lista según condición.

Ejercicio 30

Enunciado:
Crea una función `suma_dos_listas` que reciba dos listas del mismo tamaño y devuelva una lista con la suma de sus elementos por posición.

Solución:

```
def suma_dos_listas(lista1, lista2):
    return [a + b for a, b in zip(lista1, lista2)]


print(suma_dos_listas([1,2,3],[4,5,6]))
```

Explicación:

- `zip()` combina elementos de las dos listas por posición.
- List comprehension suma los pares de elementos.

Comandos usados:

- `zip()`: combina listas.
- `for a,b in zip()`: recorre elementos emparejados.
- `[]`: crea lista.
- `return`: devuelve resultado.

Ejercicio 31

Enunciado:
Crea una función `sumar_n` que reciba cualquier cantidad de números y devuelva su suma.

Solución:

```
def sumar_n(*numeros):
    return sum(numeros)


print(sumar_n(1, 2, 3, 4))
```

Explicación:

- `*numeros` permite recibir un número arbitrario de argumentos.
- `sum()` suma todos los elementos de la tupla `numeros`.

Comandos usados:

- `*args`: parámetro que acepta múltiples argumentos.
- `sum()`: suma elementos.
- `return`: devuelve resultado.

Ejercicio 32

Enunciado:
Crea una función `info_persona` que reciba nombre y edad, y devuelva un string con la información formateada.

Solución:

```
def info_persona(nombre, edad):
    return f"Nombre: {nombre}, Edad: {edad}"


print(info_persona("Ana", 25))
```

Explicación:

- Los parámetros se reciben normalmente.
- `f-string` permite insertar variables en el string.

Comandos usados:

- `f"{variable}"`: formatea string.
- `return`: devuelve resultado.

Ejercicio 33

Enunciado:
Crea una función `min_max_lista` que reciba una lista de números y devuelva una tupla con el mínimo y máximo.

Solución:

```
def min_max_lista(lista):
    return (min(lista), max(lista))


print(min_max_lista([3,7,1,9]))
```

Explicación:

- `min()` devuelve el valor mínimo de la lista.
- `max()` devuelve el máximo.
- Se devuelven ambos en una **tupla**.

Comandos usados:

- `min()`, `max()`: mínimo y máximo.
- `()` define tupla.
- `return`: devuelve resultado.

Ejercicio 34

Enunciado:
Crea una función `filtrar_pares` que reciba una lista y devuelva otra solo con los números pares.

Solución:

```
def filtrar_pares(lista):
    return [x for x in lista if x % 2 == 0]


print(filtrar_pares([1,2,3,4,5,6]))
```

Explicación:

- List comprehension filtra los elementos que cumplan `x % 2 == 0`.

Comandos usados:

- `%`: operador módulo.
- `==`: comparación.
- List comprehension: crear lista según condición.
- `return`: devuelve resultado.

Ejercicio 35

Enunciado:
Crea una función `ordenar_diccionario` que reciba un diccionario y devuelva una lista de tuplas ordenadas por valor.

Solución:

```
def ordenar_diccionario(dic):
    return sorted(dic.items(), key=lambda x: x[1])


print(ordenar_diccionario({"a":3,"b":1,"c":2}))
```

Explicación:

- `dic.items()` devuelve tuplas `(clave, valor)`.
- `sorted(..., key=...)` ordena por valor usando **lambda**.

Comandos usados:

- `dict.items()`: devuelve tuplas clave-valor.
- `sorted()`: ordena lista.
- `lambda`: función anónima.
- `return`: devuelve resultado.

Ejercicio 36

Enunciado:
Crea una función `es_capicua_num` que reciba un número y devuelva `True` si es capicúa.

Solución:

```
def es_capicua_num(n):
    n_str = str(n)
    return n_str == n_str[::-1]


print(es_capicua_num(12321))
print(es_capicua_num(12345))
```

Explicación:

- Convertimos el número a string para invertirlo.
- Comparamos la cadena original con la invertida.

Comandos usados:

- `str()`: convierte a string.
- `[::-1]`: invierte cadena.
- `==`: comparación.
- `return`: devuelve resultado.

Ejercicio 37

Enunciado:
Crea una función `diccionario_invertido` que reciba un diccionario y devuelva otro con claves y valores intercambiados.

Solución:

```
def diccionario_invertido(dic):
    return {v: k for k, v in dic.items()}


print(diccionario_invertido({"a":1,"b":2,"c":3}))
```

Explicación:

- **Dictionary comprehension** crea un nuevo diccionario con clave y valor invertidos.

Comandos usados:

- `{v:k for k,v in dic.items()}`: diccionario por comprensión.
- `dict.items()`: tuplas clave-valor.
- `return`: devuelve resultado.

Ejercicio 38

Enunciado:
Crea una función `aplicar_funcion_lista` que reciba una lista y una función, y devuelva la lista con la función aplicada a cada elemento.

Solución:

```
def aplicar_funcion_lista(lista, funcion):
    return [funcion(x) for x in lista]


print(aplicar_funcion_lista([1,2,3], lambda x: x**2))
```

Explicación:

- La función recibe otra función como parámetro.
- Se aplica a cada elemento usando list comprehension.

Comandos usados:

- `lambda`: función anónima.
- List comprehension.
- `return`: devuelve resultado.

Ejercicio 39

Enunciado:
Crea una función `sumar_diccionario` que reciba un diccionario y devuelva la suma de sus valores.

Solución:

```
def sumar_diccionario(dic):
    return sum(dic.values())


print(sumar_diccionario({"a":1,"b":2,"c":3}))
```

Explicación:

- `dic.values()` devuelve todos los valores del diccionario.
- `sum()` calcula la suma total.

Comandos usados:

- `dict.values()`: valores del diccionario.
- `sum()`: suma elementos.
- `return`: devuelve resultado.

Ejercicio 40

Enunciado:
Crea una función `recursiva_suma_n` que reciba un número `n` y devuelva la suma de todos los números desde 1 hasta `n` usando recursión.

Solución:

```
def recursiva_suma_n(n):
    if n <= 0:
        return 0
    else:
        return n + recursiva_suma_n(n - 1)


print(recursiva_suma_n(5))
```

Explicación:

- La función se llama a sí misma restando 1 hasta llegar a 0.
- Se suman todos los valores en la vuelta de la recursión.

Comandos usados:

- Recursión: función que se llama a sí misma.
- `if-else`: condicional.
- `+`: suma.
- `return`: devuelve resultado.

Ejercicio 41

Enunciado:
Crea una función `funcion_anidada` que contenga otra función dentro y que la función interna sume 10 a un número.

Solución:

```
def funcion_anidada(n):
    def sumar_diez(x):
        return x + 10
    return sumar_diez(n)

print(funcion_anidada(5))
```

Explicación:

- Una función puede definirse dentro de otra.
- La función interna `sumar_diez` se llama dentro de la función principal.

Comandos usados:

- `def`: define función.
- Función anidada: función dentro de otra.
- `return`: devuelve resultado.

Ejercicio 42

Enunciado:
Crea una función `filtrar_diccionario` que reciba un diccionario y un valor mínimo, y devuelva un diccionario con valores mayores o iguales al mínimo.

Solución:

```
def filtrar_diccionario(dic, min_valor):
    return {k:v for k,v in dic.items() if v >= min_valor}


print(filtrar_diccionario({"a":5,"b":2,"c":7}, 5))
```

Explicación:

- Se usa **dictionary comprehension** con condición.
- Devuelve un diccionario filtrado.

Comandos usados:

- `{k:v for k,v in dic.items()}`: diccionario por comprensión.
- `>=`: comparación.
- `return`: devuelve resultado.

Ejercicio 43

Enunciado:
Crea una función `aplicar_varias_funciones` que reciba un número y una lista de funciones, y devuelva una lista con los resultados.

Solución:

```
def aplicar_varias_funciones(n, funciones):
    return [f(n) for f in funciones]


print(aplicar_varias_funciones(5, [lambda x: x+1, lambda x: x*2, lambda x: x**2]))
```

Explicación:

- Se aplica cada función de la lista al número `n`.
- Se devuelve la lista de resultados.

Comandos usados:

- List comprehension.
- `lambda`: función anónima.
- `return`: devuelve resultado.

Ejercicio 44

Enunciado:
Crea una función `contar_ocurrencias` que reciba una lista de strings y devuelva un diccionario con la cantidad de veces que aparece cada palabra.

Solución:

```
def contar_ocurrencias(lista):
    resultado = {}
    for palabra in lista:
        if palabra in resultado:
            resultado[palabra] += 1
        else:
            resultado[palabra] = 1
    return resultado


print(contar_ocurrencias(["a","b","a","c","b","a"]))
```

Explicación:

- Recorre la lista y acumula en un diccionario la cantidad de ocurrencias.

Comandos usados:

- `for-in`: recorre lista.
- `in`: verifica existencia de clave.
- `+=`: suma y asigna.
- Diccionario: estructura clave-valor.
- `return`: devuelve resultado.

Ejercicio 45

Enunciado:
Crea una función `decorador_simple` que reciba otra función y la ejecute mostrando un mensaje antes y después.

Solución:

```
def decorador_simple(func):
    def nueva_funcion():
        print("Antes de la función")
        func()
        print("Después de la función")
    return nueva_funcion

def saludar():
    print("Hola!")

saludar_decorada = decorador_simple(saludar)
saludar_decorada()
```

Explicación:

- Los **decoradores** son funciones que modifican otras funciones.
- Se define `nueva_funcion` dentro de `decorador_simple` y se devuelve.

Comandos usados:

- Función como parámetro.
- Función anidada.
- `print()`: muestra texto.
- `return`: devuelve función.

Ejercicio 46

Enunciado:
Crea una función `aplanar_lista` que reciba una lista de listas y devuelva una lista plana con todos los elementos.

Solución:

```
def aplanar_lista(lista):
    return [elem for sublista in lista for elem in sublista]


print(aplanar_lista([[1,2],[3,4],[5]]))
```

Explicación:

- **List comprehension anidada** recorre cada sublista y extrae los elementos.

Comandos usados:

- List comprehension doble.
- `return`: devuelve resultado.

Ejercicio 47

Enunciado:
Crea una función `funcion_recursiva_diccionario` que reciba un número `n` y devuelva un diccionario con clave=numero y valor=factorial usando recursión.

Solución:

```
def funcion_recursiva_diccionario(n):
    if n == 0:
        return {0:1}
    else:
        dic = funcion_recursiva_diccionario(n-1)
        dic[n] = n * dic[n-1]
        return dic


print(funcion_recursiva_diccionario(5))
```

Explicación:

- Se usa recursión para construir un diccionario con factoriales desde 0 hasta `n`.

Comandos usados:

- Recursión.
- Diccionario.
- `*`: multiplicación.
- `return`: devuelve resultado.

Ejercicio 48

Enunciado:
Crea una función `filtrar_palabras` que reciba un string y devuelva una lista con palabras mayores a 3 letras.

Solución:

```
def filtrar_palabras(texto):
    return [palabra for palabra in texto.split() if len(palabra) > 3]


print(filtrar_palabras("Hola el mundo de Python"))
```

Explicación:

- `split()` divide el texto en palabras.
- List comprehension filtra por longitud de palabra.

Comandos usados:

- `split()`: separa palabras.
- `len()`: longitud.
- List comprehension.
- `return`: devuelve resultado.

Ejercicio 49

Enunciado:
Crea una función `transformar_diccionario` que reciba un diccionario y devuelva otro multiplicando sus valores por 2.

Solución:

```
def transformar_diccionario(dic):
    return {k:v*2 for k,v in dic.items()}


print(transformar_diccionario({"a":1,"b":2,"c":3}))
```

Explicación:

- Dictionary comprehension multiplica cada valor por 2.

Comandos usados:

- `{k:v for k,v in dic.items()}`: diccionario por comprensión.
- `*`: multiplicación.
- `return`: devuelve resultado.

Ejercicio 50

Enunciado:
Crea una función `aplicar_funciones_anidadas` que reciba un número y aplique varias funciones internas en secuencia, devolviendo el resultado final.

Solución:

```
def aplicar_funciones_anidadas(n):
    def sumar_dos(x):
        return x + 2
    def multiplicar_tres(x):
        return x * 3
    return multiplicar_tres(sumar_dos(n))

print(aplicar_funciones_anidadas(4))
```

Explicación:

- Se definen funciones internas y se aplican secuencialmente.
- Devuelve el resultado final de todas las operaciones.

Comandos usados:

- Funciones anidadas.
- `return`: devuelve resultado.
- `*`: multiplicación.
- `+`: suma.

MANUAL DETALLADO DE EJERCICIOS DE POO

Conceptos Fundamentales de Clases y Objetos (Ejercicios 1-3)

Ejercicio 1 – Crear una clase básica

Objetivo: Definir una clase `Persona` con atributos de instancia y crear un objeto para acceder a ellos. **Implementación y Explicación:**

1. Se define la clase usando `class Persona:`.
2. Se utiliza el **constructor** `__init__(self, nombre, edad)` para inicializar los atributos `nombre` y `edad`.
3. La palabra clave **`self`** hace referencia al propio objeto que se está creando.
4. Se crea un objeto de la clase mediante `persona1 = Persona("Ana", 25)`.
5. Se accede a los atributos del objeto utilizando la notación de punto, por ejemplo, `print(persona1.nombre)`. **Controles y Comandos de POO:** `class`, `__init__`, `self`, creación de objetos, atributos.

Ejercicio 2 – Método dentro de una clase

Objetivo: Añadir un método (`saludar`) a la clase para interactuar con los atributos del objeto. **Implementación y Explicación:**

1. Se define un método dentro de la clase usando `def saludar(self):`.
2. El método utiliza `self.nombre` para acceder al atributo `nombre` del objeto.
3. Se invoca el método desde el objeto creado usando `persona1.saludar()`. **Controles y Comandos de POO:** Métodos, uso de `self`, llamada a métodos desde objetos.

Ejercicio 3 – Encapsulamiento básico

Objetivo: Hacer que un atributo (`edad`) sea privado y solo accesible mediante un método público. **Implementación y Explicación:**

1. Se declara el atributo de instancia como privado usando el doble guion bajo: `self.__edad = edad`.
2. Este uso de `__edad` convierte el atributo en privado (encapsulamiento).
3. Se crea un **método público** (`mostrar_edad`) que es el único que puede acceder y mostrar el valor de `__edad`. **Controles y Comandos de POO:** Encapsulamiento, atributos privados, métodos públicos.

Herencia y Polimorfismo (Ejercicios 4-7)

Ejercicio 4 – Herencia simple

Objetivo: Crear una clase hija (`Estudiante`) que herede de una clase base (`Persona`). **Implementación y Explicación:**

1. La herencia se define en la declaración de la clase: `class Estudiante(Persona):`.

2. En el constructor de la clase hija, se utiliza `super().__init__(nombre, edad)` para llamar al constructor de la clase base (`Persona`) y heredar sus atributos iniciales.
3. Se añade el atributo específico de la clase hija, `self.curso`.
4. El objeto `estudiante1` puede acceder a atributos heredados (`estudiante1.nombre`) y propios (`estudiante1.curso`). **Controles y Comandos de POO:** Herencia, `super()`, atributos adicionales.

Ejercicio 5 – Polimorfismo básico

Objetivo: Redefinir un método en la clase hija para que presente un comportamiento distinto al de la clase base. **Implementación y Explicación:**

1. Se define el método `presentarse(self)` tanto en la clase `Persona` como en la clase `Estudiante`.
2. La redefinición del método en la clase hija es lo que demuestra el **polimorfismo**.
3. Al llamar a `persona1.presentarse()` y `estudiante1.presentarse()`, cada objeto ejecuta la versión del método correspondiente a su clase. **Controles y Comandos de POO:** Polimorfismo, redefinición de métodos.

Ejercicio 6 – Clase con lista de objetos (Composición)

Objetivo: Crear una clase (`Curso`) que contenga una lista de objetos de otra clase (`Estudiante`). **Implementación y Explicación:**

1. Dentro del constructor de `Curso`, se inicializa una lista vacía: `self.alumnos = []`.
2. El método `agregar_estudiante(self, estudiante)` usa el comando `append()` para agregar objetos `Estudiante` a la lista.
3. El método `mostrar_estudiantes` utiliza un bucle `for e in self.alumnos:` para recorrer y acceder a los atributos de cada objeto dentro de la lista (ejemplo: `print(e.nombre)`). **Controles y Comandos de POO:** Composición, listas de objetos, métodos de agregación y recorrido (`append()`, `for...in`).

Ejercicio 7 – Métodos especiales: __str__

Objetivo: Definir una representación legible de un objeto para la función `print()`.
Implementación y Explicación:

1. Se implementa el método especial `__str__(self)`.
2. Este método debe **devolver** una cadena de texto (string) que represente el objeto de manera clara, por ejemplo: `return f"{self.nombre} tiene {self.edad} años"`.
3. Al usar `print(persona1)`, se llama automáticamente al método `__str__` definido. **Controles y Comandos de POO:** Métodos especiales, representación de objetos (`__str__`).

Encapsulamiento Avanzado y Control de Atributos (Ejercicios 8, 18)

Ejercicio 8 – Encapsulamiento con getters y setters

Objetivo: Permitir el acceso y la modificación controlada de un atributo privado. **Implementación y Explicación:**

1. Se mantiene el atributo `__edad` como privado.
2. Se define un método **getter** (`get_edad`) para retornar el valor del atributo privado.
3. Se define un método **setter** (`set_edad`) que incluye un control (`if nueva_edad > 0:`) antes de modificar el valor de `self.__edad`. **Controles y Comandos de POO:** Getters y setters, validación de atributos.

Ejercicio 18 – Encapsulamiento con validación

Objetivo: Aplicar una validación estricta en el método setter para asegurar que el atributo privado cumpla ciertas condiciones. **Implementación y Explicación:**

1. El método `set_edad(self, edad)` comprueba que la edad esté dentro de un rango lógico (`0 < edad < 120`).
2. Si la validación no se cumple (ej. al intentar `p.set_edad(130)`), el atributo privado `__edad` no se modifica.
3. El acceso para lectura sigue siendo a través del `get_edad()`. **Controles y Comandos de POO:** Encapsulamiento, validación en setters.

Clases Abstractas y Métodos Especiales de Clase/Instancia (Ejercicios 9, 13, 14, 15, 17)

Ejercicio 9 – Clase abstracta

Objetivo: Obligar a las subclases a implementar ciertos métodos, haciendo la clase base abstracta. **Implementación y Explicación:**

1. Se importan `ABC` (Abstract Base Class) y `abstractmethod`.
2. La clase base se define como abstracta: `class Animal(ABC):`.
3. El método a ser obligatorio se decora con `@abstractmethod` (`def hacer_sonido(self): pass`).
4. La clase hija (`Perro`) debe implementar `hacer_sonido`, o la instanciación fallará. **Controles y Comandos de POO:** Clases abstractas (`ABC`), métodos abstractos (`@abstractmethod`), herencia obligatoria.

Ejercicio 13 – Método de clase

Objetivo: Crear un método que opere sobre la clase misma y no sobre una instancia individual. **Implementación y Explicación:**

1. Se usa un atributo de clase compartido (`total = 0`).
2. Se aplica el decorador `@classmethod` al método (`incrementar`).

3. El primer argumento es `cls` (referencia a la clase), que se usa para modificar el atributo de clase (`cls.total += 1`).
4. Se llama directamente desde la clase: `Contador.incrementar()`. **Controles y Comandos de POO:** Métodos de clase, atributos de clase, `@classmethod`.

Ejercicio 14 – Método estático

Objetivo: Crear un método que no necesita acceder ni al objeto (`self`) ni a la clase (`cls`).
Implementación y Explicación:

1. Se usa el decorador `@staticmethod`.
2. El método (`sumar(a, b)`) solo utiliza los argumentos pasados.
3. Se llama directamente desde la clase, sin necesidad de crear un objeto: `print(Matematica.sumar(5, 7))`. **Controles y Comandos de POO:** Métodos estáticos, decoradores, retorno de valores.

Ejercicio 15 – Propiedades (`@property`)

Objetivo: Permitir que un método que calcula un valor sea accedido como si fuera un atributo.
Implementación y Explicación:

1. Se define el método `area(self)` y se decora con `@property`.
2. El cálculo (`self.base * self.altura`) se realiza internamente.
3. Se accede al valor sin usar paréntesis: `print(r.area)`. **Controles y Comandos de POO:** Propiedades, `@property`, cálculo de atributos, encapsulamiento.

Ejercicio 17 – Sobrecarga de operadores (`__add__`)

Objetivo: Redefinir cómo funciona el operador `+` para sumar objetos personalizados.
Implementación y Explicación:

1. Se implementa el método mágico `__add__(self, otro)`.
2. El método retorna un **nuevo objeto** (`Vector`) cuyos atributos son la suma de los atributos de los objetos originales.
3. La operación `v3 = v1 + v2` ejecuta la sobrecarga. **Controles y Comandos de POO:** Métodos especiales, sobrecarga de operadores (`__add__`), creación de nuevos objetos.

Conceptos Avanzados de Herencia y Diseño (Ejercicios 10, 11, 12, 16, 19, 20, 21)

Ejercicio 10 – Composición de clases

Objetivo: Incluir un objeto de una clase como atributo dentro de otra clase. **Implementación y Explicación:**

1. Se define la clase `Motor`.
2. La clase `Coche` recibe un objeto `motor` en su constructor y lo almacena: `self.motor = motor`.

3. Para acceder a los atributos del objeto contenido, se encadenan los puntos: `coche1.motor.tipo`. **Controles y Comandos de POO:** Composición, acceso a atributos de objetos anidados.

Ejercicio 11 – Herencia múltiple

Objetivo: Crear una clase que herede atributos y métodos de dos o más clases base. **Implementación y Explicación:**

1. Se define la clase hija especificando múltiples bases: `class C(A, B):`.
2. El objeto de la clase C hereda y puede ejecutar los métodos de ambas clases base (`c.metodo_a()`, `c.metodo_b()`). **Controles y Comandos de POO:** Herencia múltiple, llamada a métodos de clases base.

Ejercicio 12 – Sobrescribir constructor

Objetivo: Crear un constructor en la clase hija que extienda el comportamiento del constructor base. **Implementación y Explicación:**

1. En el constructor de `Estudiante`, se llama explícitamente al constructor de la clase superior usando `super().__init__(nombre)`.
2. Esto inicializa los atributos heredados y permite que la clase hija defina su propio atributo adicional (`self.curso`). **Controles y Comandos de POO:** Constructor en herencia, `super().__init__`, atributos adicionales.

Ejercicio 16 – Setter de propiedad (`@setter`)

Objetivo: Definir cómo se debe modificar una propiedad calculada o controlada. **Implementación y Explicación:**

1. Se define primero el `@property` (por ejemplo, `area`).
2. Luego, se define el método **setter** usando el nombre de la propiedad y el decorador `@area.setter`.
3. El setter permite realizar cálculos inversos o validaciones para actualizar los atributos base, como calcular la nueva base basado en el valor de área deseado (`self._base = valor / self._altura`).
4. Se modifica la propiedad como si fuera un atributo: `r.area = 15`. **Controles y Comandos de POO:** Getter y setter con `@property` y `@setter`.

Ejercicio 19 – Herencia y super para extensión

Objetivo: Usar `super()` para ejecutar el método de la clase base y luego añadir funcionalidad extra. **Implementación y Explicación:**

1. La clase `Estudiante` redefine el método `saludar`.
2. Dentro del método redefinido, `super().saludar()` se usa para llamar y ejecutar la implementación de `Persona.saludar()`.
3. Posteriormente, se agrega la funcionalidad específica de `Estudiante`. **Controles y Comandos de POO:** Herencia, `super()`, extensión de métodos.

Ejercicio 20 – Composición avanzada (Anidamiento)

Objetivo: Modelar una relación "tiene un" donde una clase almacena un objeto de otra y utiliza sus atributos. **Implementación y Explicación:**

1. La clase `Libro` contiene un objeto `Autor`.
2. El método `mostrar` accede al atributo `nombre` a través del objeto `autor` almacenado: `self.autor.nombre`. **Controles y Comandos de POO:** Composición, acceso a atributos de objetos anidados, métodos de clase.

Ejercicio 21 – Herencia múltiple con métodos iguales (MRO)

Objetivo: Entender el orden de resolución de métodos (MRO) en la herencia múltiple.
Implementación y Explicación:

1. Las clases `A` y `B` tienen métodos idénticos (`saludo`).
2. La clase `C` hereda de ambas: `class C(A, B):`.
3. Al llamar a `c.saludo()`, Python sigue el **MRO** y ejecuta el método de la primera clase listada en la herencia, que es `A`. **Controles y Comandos de POO:** Herencia múltiple, MRO (Method Resolution Order), resolución de métodos.

Encapsulamiento Detallado y Atributos de Clase (Ejercicios 22, 23, 24, 25)

Ejercicio 22 – Método privado

Objetivo: Crear un método que solo pueda ser invocado internamente dentro de la clase.
Implementación y Explicación:

1. Se define el método como privado usando doble guion bajo: `def __metodo_privado(self):`.
2. Para ejecutarlo, se crea un método público (`llamar_privado`) que internamente llama a `self.__metodo_privado()`. **Controles y Comandos de POO:** Métodos privados, encapsulamiento.

Ejercicio 23 – Clase con contador de instancias

Objetivo: Utilizar un atributo compartido para contar cuántos objetos se han creado.
Implementación y Explicación:

1. Se define `total = 0` como un **atributo de clase**, accesible por todos los objetos de `Persona`.
2. Dentro del constructor `__init__`, se incrementa este contador usando el nombre de la clase: `Persona.total += 1`. **Controles y Comandos de POO:** Atributos de clase, incremento en constructor.

Ejercicio 24 – Herencia y método extendido (Polimorfismo)

Objetivo: Ilustrar la redefinición completa de un método en la clase hija. **Implementación y Explicación:**

1. La clase `Estudiante` implementa una versión diferente de `presentarse` que la clase `Persona`.
2. Al llamar al método desde el objeto `e` (Estudiante), se ejecuta la versión redefinida (Polimorfismo). **Controles y Comandos de POO:** Herencia, redefinición de métodos, polimorfismo.

Ejercicio 25 – Método estático con validación

Objetivo: Usar un método estático para realizar una operación de utilidad sin depender del estado del objeto. **Implementación y Explicación:**

1. Se usa `@staticmethod` para la función `es_par(n)`.
2. La función utiliza el operador módulo (`%`) para comprobar si el número es divisible por 2 (`n % 2 == 0`) y retorna un valor booleano.
3. Se llama directamente a través de la clase: `Utilidades.es_par(10)`. **Controles y Comandos de POO:** Métodos estáticos, operadores aritméticos, retorno de valores.

Sobrecarga de Operadores y Estructuras de Datos Complejas (Ejercicios 26, 27, 29)

Ejercicio 26 – Sobrecarga de operadores con strings

Objetivo: Redefinir `+` para combinar atributos de string y crear un nuevo objeto. **Implementación y Explicación:**

1. El método `__add__` concatena los atributos `nombre` de `self` y `otra`.
2. Retorna una **nueva instancia** de `Persona` con el nombre combinado: `return Persona(self.nombre + " " + otra.nombre)`. **Controles y Comandos de POO:** Sobrecarga de operadores (`__add__`), creación de objetos en operadores.

Ejercicio 27 – Clase con lista de objetos y métodos de filtrado

Objetivo: Usar la composición (objetos dentro de objetos) junto con lógica de filtrado. **Implementación y Explicación:**

1. La clase `Curso` almacena objetos `Estudiante` en `self.alumnos`.
2. El método `mayores_de_edad()` usa una **comprensión de listas** para iterar sobre `self.alumnos` y filtrar aquellos donde `e.edad >= 18`. **Controles y Comandos de POO:** Composición, listas de objetos, comprensión de listas.

Ejercicio 29 – Polimorfismo con lista de objetos

Objetivo: Demostrar que una lista que contiene diferentes tipos de objetos puede ejecutar el mismo método con resultados distintos. **Implementación y Explicación:**

1. Se define una lista `animales` que contiene instancias de `Perro` y `Gato` (subclases de `Animal`).

2. El bucle `for a in animales: a.hacer_sonido()` ejecuta la implementación específica de `hacer_sonido` para cada clase, demostrando el polimorfismo. **Controles y Comandos de POO:** Polimorfismo, listas de objetos, herencia.

Decoradores, Excepciones y Métodos de Clase (Ejercicios 28, 30, 31, 32)

Ejercicio 28 – Método de clase y atributos de clase

Objetivo: Usar un método de clase para reportar el valor de un atributo de clase (contador).
Implementación y Explicación:

1. `total` es un atributo de clase que se incrementa en `__init__`.
2. El método `contar()` usa el decorador `@classmethod` y recibe `cls` como argumento.
3. Retorna el valor de `cls.total`, que es el número total de instancias creadas. **Controles y Comandos de POO:** Métodos de clase, atributos de clase, `@classmethod`, contador de instancias.

Ejercicio 30 – Decorador de método

Objetivo: Envolver un método con una función externa para añadir funcionalidad (como logs) antes de su ejecución. **Implementación y Explicación:**

1. Se define la función `decorador(func)` que devuelve una función `wrapper`.
2. La `wrapper` ejecuta código extra (ej. `print("Ejecutando método...")`) y luego llama a la función original (`func(*args, **kwargs)`).
3. Se aplica el decorador al método `saludar` usando `@decorador`. **Controles y Comandos de POO:** Decoradores de métodos, funciones internas, `*args` y `**kwargs`.

Ejercicio 31 – Manejo de excepciones en métodos

Objetivo: Controlar el flujo de ejecución lanzando una excepción si se incumplen las condiciones.
Implementación y Explicación:

1. En el método `retirar(self, monto)`, se valida si `monto > self.saldo`.
2. Si la condición de error se cumple, se utiliza el comando `raise ValueError("Saldo insuficiente")` para detener la ejecución del método y señalar el error. **Controles y Comandos de POO:** Excepciones (`raise ValueError`), métodos, atributos, control de errores.

Ejercicio 32 – Herencia y excepciones

Objetivo: Sobrescribir un método de la clase base y añadir una validación específica de la subclase antes de ejecutar la lógica original. **Implementación y Explicación:**

1. `CuentaAhorros` redefine `retirar`.
2. Primero, verifica si el retiro excede un límite específico (`if monto > 500:`).
3. Si la validación local pasa, llama al método de la clase base para la lógica principal de retiro: `super().retirar(monto)`. **Controles y Comandos de POO:** Herencia, sobrescritura de métodos, excepciones, `super()`.

Mixins, Abstracción Avanzada y Propiedades Rigurosas (Ejercicios 33-38)

Ejercicio 33 – Mixins

Objetivo: Usar una clase (`InfoMixin`) para inyectar funcionalidad (métodos) a otras clases sin que haya una relación jerárquica principal. **Implementación y Explicación:**

1. `InfoMixin` define `mostrar_info`.
2. Este método utiliza `self.__class__.__name__` para obtener el nombre de la clase hija y `self.__dict__` para acceder a todos los atributos del objeto.
3. `Persona` hereda de `InfoMixin` para obtener el método `mostrar_info()`. **Controles y Comandos de POO:** Mixins, herencia, atributos de objetos (`__dict__`).

Ejercicio 34 – Herencia múltiple con mixins

Objetivo: Combinar una clase funcional principal con un Mixin para añadir funcionalidad transversal. **Implementación y Explicación:**

1. `EstudianteInfo(Estudiante, InfoMixin)` hereda los atributos de `Estudiante` y los métodos de `InfoMixin`.
2. Permite que un objeto de `EstudianteInfo` ejecute el método `mostrar_info()` provisto por el Mixin. **Controles y Comandos de POO:** Herencia múltiple, mixins, combinación de métodos.

Ejercicio 35 – Encapsulamiento avanzado con propiedades

Objetivo: Usar `@property` y `@setter` para aplicar validación compleja a un atributo privado o semi-privado. **Implementación y Explicación:**

1. El atributo se almacena como `self._edad` (convención semi-privada).
2. El `@property def edad` actúa como el getter.
3. El `@edad.setter` aplica una validación estricta (`if 0 < valor < 120:`). Si el valor es inválido, imprime un mensaje y no modifica el atributo. **Controles y Comandos de POO:** Propiedades, encapsulamiento, validación de datos.

Ejercicio 36 – Sobrecarga de operadores con objetos complejos (`__mul__`)

Objetivo: Redefinir el operador de multiplicación (`*`) para realizar operaciones específicas entre un objeto y un escalar. **Implementación y Explicación:**

1. Se implementa el método `__mul__(self, escalar)`.
2. La operación `v * 5` ejecuta este método, multiplicando cada componente (`self.x * escalar`, `self.y * escalar`).
3. Retorna un nuevo objeto `Vector` con los resultados. **Controles y Comandos de POO:** Sobrecarga de operadores (`__mul__`), objetos inmutables (creación de nuevo objeto).

Ejercicio 37 – Polimorfismo con múltiples subclases

Objetivo: Demostrar cómo una jerarquía de herencia con múltiples clases derivadas maneja el mismo método de manera única. **Implementación y Explicación:**

1. Las subclases `Perro`, `Gato`, y `Vaca` heredan de `Animal` y redefinen `hacer_sonido()`.
2. Una lista heterogénea de estas subclases es iterada, y cada objeto responde a `a.hacer_sonido()` de forma polimórfica. **Controles y Comandos de POO:** Polimorfismo, herencia, listas de objetos.

Ejercicio 38 – Clase abstracta con múltiples métodos obligatorios

Objetivo: Asegurar que las clases concretas implementen un conjunto completo de métodos requeridos por la base abstracta. **Implementación y Explicación:**

1. La clase `Figura(ABC)` define dos métodos con `@abstractmethod`: `area()` y `perimetro()`.
2. La subclase `Rectangulo` está obligada a implementar **ambos** métodos para poder ser instanciada. **Controles y Comandos de POO:** Clases abstractas, métodos abstractos, herencia obligatoria.

Métodos Mágicos, Combinación de Clases y Patrones de Diseño (Ejercicios 39-42)

Ejercicio 39 – Métodos mágicos __eq__ y __repr__

Objetivo: Controlar cómo se comparan los objetos (==) y cómo se representan para desarrolladores. **Implementación y Explicación:**

1. `__eq__(self, otra)` redefine el operador de igualdad (==) para comparar objetos basados en un atributo (ej. `self.nombre == otra.nombre`).
2. `__repr__(self)` define una representación del objeto (usualmente para debugging), que se usa cuando se imprime el objeto directamente si no existe `__str__` o para mostrar estructuras de datos. **Controles y Comandos de POO:** Métodos mágicos (`__eq__`, `__repr__`), sobrecarga de operadores, representación de objetos.

Ejercicio 40 – Clase con métodos de clase y estáticos

Objetivo: Combinar `@staticmethod` (para lógica pura) y `@classmethod` (para gestionar el estado de la clase). **Implementación y Explicación:**

1. `celsius_a_fahrenheit` es `@staticmethod`; realiza la conversión matemática.
2. Este método estático accede y modifica el atributo de clase `Temperatura.conversiones += 1`.
3. `contar_conversiones` es un `@classmethod` que usa `cls` para devolver el estado compartido (`cls.conversiones`). **Controles y Comandos de POO:** Métodos estáticos y de clase, atributos de clase, contador global de objetos.

Ejercicio 41 – Patrón Singleton

Objetivo: Asegurar que una clase solo pueda tener una única instancia en toda la aplicación. **Implementación y Explicación:**

1. Se define un atributo de clase privado `__instancia = None`.
2. Se sobrescribe el método especial `__new__`, que es el que se encarga de crear la instancia antes que `__init__` la inicialice.
3. Dentro de `__new__`, si `__instancia` es `None`, se crea la instancia usando `super().__new__(cls)` y se almacena; si ya existe, se devuelve la instancia existente.
4. El resultado es que `bd1 is bd2` es `True`. **Controles y Comandos de POO:** `__new__`, atributos privados de clase, control de instanciación.

Ejercicio 42 – Factory Method (Método de Fábrica)

Objetivo: Crear un método o función que decida qué tipo de objeto crear en función de un parámetro de entrada, delegando la creación de la instancia. **Implementación y Explicación:**

1. Se utiliza una función externa (`crear_animal(tipo)`) que recibe un parámetro.
2. Mediante condicionales (`if/elif`), la función retorna una instancia de la subclase apropiada (`Perro()` o `Gato()`).
3. Esto abstrae la lógica de instanciación del código cliente. **Controles y Comandos de POO:** Factory pattern, herencia, polimorfismo.

Composición Profunda y Utilidades (Ejercicios 43-46)

Ejercicio 43 – Composición profunda

Objetivo: Modelar estructuras jerárquicas complejas donde los objetos están anidados varios niveles (Empresa -> Departamento -> Empleado). **Implementación y Explicación:**

1. `Empresa` almacena una lista de objetos `Departamento`.
2. Cada `Departamento` almacena una lista de objetos `Empleado`.
3. Para acceder a los empleados, se requieren **bucles anidados** que primero recorren los departamentos y luego los empleados de cada departamento. **Controles y Comandos de POO:** Composición, listas de objetos, bucles anidados.

Ejercicio 44 – Mixin avanzado (JSON)

Objetivo: Añadir una funcionalidad utilitaria de serialización a cualquier clase que herede el Mixin. **Implementación y Explicación:**

1. Se importa la librería `json`.
2. `JsonMixin` define `to_json(self)`.
3. Utiliza `json.dumps(self.__dict__)` para convertir el diccionario de atributos del objeto (`__dict__`) en una cadena de formato JSON. **Controles y Comandos de POO:** Mixins, herencia, serialización de objetos (`json.dumps`, `__dict__`).

Ejercicio 45 – Propiedades con validación compleja

Objetivo: Asegurar la integridad de un atributo de forma estricta usando setters de propiedades. **Implementación y Explicación:**

1. El `@precio.setter` valida que el `valor` sea mayor que cero (`if valor > 0:`).
2. Si el valor es inválido (ej. negativo), se imprime un mensaje y se previene la modificación del atributo `_precio`. **Controles y Comandos de POO:** Propiedades, validación, encapsulamiento (`@property`, `@setter`).

Ejercicio 46 – Sobrecarga de operadores con comparación (`__gt__`)

Objetivo: Redefinir un operador de comparación (mayor que, `>`) para basarse en atributos internos del objeto. **Implementación y Explicación:**

1. Se implementa el método mágico `__gt__(self, otra)` (Greater Than).
2. Este método retorna un valor booleano (`True` o `False`) basado en la comparación de los atributos: `return self.edad > otra.edad`. **Controles y Comandos de POO:** Sobrecarga de operadores (`__gt__`), comparación de objetos.

Polimorfismo y Combinación de Componentes Finales (Ejercicios 47-50)

Ejercicio 47 – Herencia compleja con métodos específicos

Objetivo: Mostrar cómo las subclases pueden heredar métodos comunes (`arrancar`) y definir su propia funcionalidad única. **Implementación y Explicación:**

1. `Coche` hereda de `Vehiculo` y añade `abrir_maletero()`.
2. `Moto` hereda de `Vehiculo` y añade `hacer_caballito()`.
3. Los objetos acceden tanto a los métodos heredados (`c.arrancar()`) como a sus métodos específicos (`c.abrir_maletero()`). **Controles y Comandos de POO:** Herencia, métodos propios y heredados.

Ejercicio 48 – Polimorfismo con listas heterogéneas

Objetivo: Demostrar el polimorfismo dinámico al tratar objetos de diferentes clases derivadas de la misma base de manera uniforme. **Implementación y Explicación:**

1. Se define una lista `vehiculos` que contiene instancias de `Coche` y `Moto`.
2. Aunque se itera sobre ellos como `Vehiculo`, al llamar a `v.arrancar()`, se ejecuta la versión específica de cada subclase. **Controles y Comandos de POO:** Polimorfismo, listas de objetos, herencia.

Ejercicio 49 – Métodos mágicos combinados (`__add__` y `__str__`)

Objetivo: Utilizar la sobrecarga de operadores para la lógica interna y el método de representación para el formato de salida. **Implementación y Explicación:**

1. `__add__` suma los atributos de `cantidad` de dos objetos `Dinero` y retorna una nueva instancia.

2. `__str__` define cómo debe mostrarse el objeto cuando se imprime, añadiendo formato (ej. el símbolo `$`).
3. `print(d3)` invoca a `__str__` automáticamente después de que `d1 + d2` invoca a `__add__`. **Controles y Comandos de POO:** Métodos mágicos (`__add__`, `__str__`), sobrecarga de operadores, representación de objetos.

Ejercicio 50 – Composición + mixins + propiedades

Objetivo: Integrar la composición de objetos, las propiedades calculadas y la funcionalidad inyectada por Mixins en una única clase. **Implementación y Explicación:**

1. **Composición:** `Pedido` contiene una lista de objetos `Producto`.
2. **Propiedad:** El `@property def total` calcula dinámicamente el total sumando los precios de todos los productos en la lista, usando comprensión de listas y `sum()`.
3. **Mixin:** La clase `Pedido` hereda de `JsonMixin` para obtener la capacidad de serializar su contenido a JSON (`pedido.to_json()`). **Controles y Comandos de POO:** Composición profunda, mixins, propiedades, métodos de clase, listas de objetos.

Ejercicio 1 – Crear una clase básica

Enunciado: Crea una clase `Persona` con atributos `nombre` y `edad`. Luego, crea un objeto y muestra sus atributos.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

persona1 = Persona("Ana", 25)
print(persona1.nombre)
print(persona1.edad)
```

Explicación:

- `class Persona:` define una clase llamada `Persona`.
- `__init__` es el **constructor** que inicializa los atributos del objeto.
- `self` hace referencia al propio objeto.
- `persona1 = Persona("Ana", 25)` crea un objeto de la clase.
- `print(persona1.nombre)` accede al atributo `nombre`.

Comandos de POO explicados: `class`, `__init__`, `self`, creación de objetos, atributos.

Ejercicio 2 – Método dentro de una clase

Enunciado: Añade un método `saludar` que imprima un saludo usando el nombre de la persona.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def saludar(self):
        print(f"Hola, me llamo {self.nombre}")

persona1 = Persona("Ana", 25)
persona1.saludar()
```

Explicación:

- `def saludar(self):` define un **método** de la clase.
- `self.nombre` accede al atributo del objeto.
- `persona1.saludar()` llama al método.

Comandos de POO explicados: Métodos, uso de `self`, llamada a métodos desde objetos.

Ejercicio 3 – Encapsulamiento básico

Enunciado: Haz que la edad sea privada y añade un método para mostrarla.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.__edad = edad  # atributo privado

    def mostrar_edad(self):
        print(f"Tengo {self.__edad} años")

persona1 = Persona("Ana", 25)
persona1.mostrar_edad()
```

Explicación:

- `__edad` convierte el atributo en **privado**.
- Solo se puede acceder mediante un **método público** como `mostrar_edad`.

Comandos de POO explicados: Encapsulamiento, atributos privados, métodos públicos.

Ejercicio 4 – Herencia simple

Enunciado: Crea una clase `Estudiante` que herede de `Persona` y tenga un atributo adicional `curso`.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

class Estudiante(Persona):
    def __init__(self, nombre, edad, curso):
        super().__init__(nombre, edad)
        self.curso = curso

estudiante1 = Estudiante("Luis", 20, "Matemáticas")
print(estudiante1.nombre, estudiante1.curso)
```

Explicación:

- `class Estudiante(Persona):` indica que `Estudiante` **hereda** de `Persona`.
- `super().__init__(nombre, edad)` llama al constructor de la clase base.
- `estudiante1.curso` es el atributo específico de `Estudiante`.

Comandos de POO explicados: Herencia, `super()`, atributos adicionales.

Ejercicio 5 – Polimorfismo básico

Enunciado: Crea un método `presentarse` que sea distinto para `Persona` y `Estudiante`.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def presentarse(self):
        print(f"Soy {self.nombre}")

class Estudiante(Persona):
    def __init__(self, nombre, edad, curso):
        super().__init__(nombre, edad)
        self.curso = curso

    def presentarse(self):
        print(f"Soy {self.nombre} y estudio {self.curso}")

persona1 = Persona("Ana", 25)
estudiante1 = Estudiante("Luis", 20, "Matemáticas")

persona1.presentarse()
estudiante1.presentarse()
```

Explicación:

- El método `presentarse` **se redefine** en la clase hija (`Estudiante`), mostrando **polimorfismo**.

Comandos de POO explicados: Polimorfismo, redefinición de métodos.

Ejercicio 6 – Clase con lista de objetos

Enunciado: Crea una clase `Curso` que tenga una lista de estudiantes y un método para mostrarlos.

```
class Estudiante:
    def __init__(self, nombre):
        self.nombre = nombre

class Curso:
    def __init__(self):
        self.alumnos = []

    def agregar_estudiante(self, estudiante):
        self.alumnos.append(estudiante)

    def mostrar_estudiantes(self):
        for e in self.alumnos:
            print(e.nombre)

curso = Curso()
curso.agregar_estudiante(Estudiante("Ana"))
curso.agregar_estudiante(Estudiante("Luis"))
curso.mostrar_estudiantes()
```

Explicación:

- `self.alumnos = []` crea una lista de objetos.
- `append()` agrega objetos a la lista.
- `for e in self.alumnos:` recorre los objetos.

Comandos de POO explicados: Composición, listas de objetos, métodos de agregación y recorrido.

Ejercicio 7 – Métodos especiales: `__str__`

Enunciado: Define un método especial `__str__` para mostrar los datos de la persona de manera bonita.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __str__(self):
        return f"{self.nombre} tiene {self.edad} años"

persona1 = Persona("Ana", 25)
print(persona1)
```

Explicación:

- `__str__` devuelve la representación **legible** del objeto.
- `print(persona1)` llama automáticamente a `__str__`.

Comandos de POO explicados: Métodos especiales, representación de objetos.

Ejercicio 8 – Encapsulamiento con getters y setters

Enunciado: Haz que la edad sea privada y permite acceder y modificar con métodos.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.__edad = edad

    def get_edad(self):
        return self.__edad

    def set_edad(self, nueva_edad):
        if nueva_edad > 0:
            self.__edad = nueva_edad

persona1 = Persona("Ana", 25)
print(persona1.get_edad())
persona1.set_edad(26)
print(persona1.get_edad())
```

Explicación:

- `get_edad` y `set_edad` permiten **acceder y modificar** un atributo privado con control.

Comandos de POO explicados: Getters y setters, validación de atributos.

Ejercicio 9 – Clase abstracta

Enunciado: Crea una clase abstracta `Animal` y obliga a implementar el método `hacer_sonido`.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def hacer_sonido(self):
        pass

class Perro(Animal):
    def hacer_sonido(self):
        print("Guau")

perro = Perro()
perro.hacer_sonido()
```

Explicación:

- `ABC` indica **clase abstracta**.
- `@abstractmethod` obliga a las subclases a implementar el método.

Comandos de POO explicados: Clases abstractas, métodos abstractos, herencia obligatoria.

Ejercicio 10 – Composición de clases

Enunciado: Una clase `Motor` dentro de un `Coche`.

```
class Motor:
    def __init__(self, tipo):
        self.tipo = tipo

class Coche:
    def __init__(self, marca, motor):
        self.marca = marca
        self.motor = motor

motor1 = Motor("V8")
coche1 = Coche("Ford", motor1)
print(f"{coche1.marca} con motor {coche1.motor.tipo}")
```

Explicación:

- La clase `Coche` **contiene** un objeto `Motor`.
- `coche1.motor.tipo` accede al atributo del objeto contenido.

Comandos de POO explicados: Composición, acceso a atributos de objetos anidados.

Ejercicio 11 – Herencia múltiple

Enunciado: Crea una clase `A` con un método y una clase `B` con otro. Luego crea `C` que herede de ambas.

```
class A:
    def metodo_a(self):
        print("Método de A")

class B:
    def metodo_b(self):
        print("Método de B")

class C(A, B):
    pass

c = C()
c.metodo_a()
c.metodo_b()
```

Explicación:

- `class C(A, B):` indica **herencia múltiple**.
- `C` hereda los métodos de `A` y `B`.

Comandos de POO explicados: Herencia múltiple, llamada a métodos de clases base.

Ejercicio 12 – Sobrescribir constructor

Enunciado: Sobrescribe el constructor de una clase hija y añade un atributo extra.

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre

class Estudiante(Persona):
    def __init__(self, nombre, curso):
        super().__init__(nombre)
        self.curso = curso

est = Estudiante("Ana", "Matemáticas")
print(est.nombre, est.curso)
```

Explicación:

- `super().__init__(nombre)` llama al constructor de la clase base.
- Se añade un nuevo atributo `curso`.

Comandos de POO explicados: Constructor en herencia, atributos adicionales.

Ejercicio 13 – Método de clase

Enunciado: Crea un método que afecte a la clase, no al objeto individual.

```
class Contador:
    total = 0

    @classmethod
    def incrementar(cls):
        cls.total += 1

Contador.incrementar()
Contador.incrementar()
print(Contador.total)
```

Explicación:

- `@classmethod` indica que el método recibe la **clase** como primer argumento (`cls`).
- Se modifica un atributo compartido por todos los objetos.

Comandos de POO explicados: Métodos de clase, atributos de clase, `@classmethod`.

Ejercicio 14 – Método estático

Enunciado: Crea un método que no dependa ni de la clase ni del objeto.

```
class Matematica:
    @staticmethod
    def sumar(a, b):
        return a + b

print(Matematica.sumar(5, 7))
```

Explicación:

- `@staticmethod` define un método que **no usa ni `self` ni `cls`**.
- Se puede llamar directamente desde la clase.

Comandos de POO explicados: Métodos estáticos, decoradores, retorno de valores.

Ejercicio 15 – Propiedades

Enunciado: Usa `@property` para acceder a un atributo calculado.

```
class Rectangulo:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    @property
    def area(self):
        return self.base * self.altura

r = Rectangulo(5, 3)
print(r.area)
```

Explicación:

- `@property` permite acceder a un **método como si fuera un atributo**.
- `r.area` devuelve el área sin llamar explícitamente a un método.

Comandos de POO explicados: Propiedades, cálculo de atributos, encapsulamiento.

Ejercicio 16 – Setter de propiedad

Enunciado: Permite modificar un atributo usando `@setter`.

```
class Rectangulo:
    def __init__(self, base, altura):
        self._base = base
        self._altura = altura

    @property
    def area(self):
        return self._base * self._altura

    @area.setter
    def area(self, valor):
        self._base = valor / self._altura

r = Rectangulo(5, 3)
print(r.area)
r.area = 15
print(r._base)
```

Explicación:

- `@area.setter` permite modificar la propiedad de forma controlada.
- Calcula la nueva base según el valor deseado del área.

Comandos de POO explicados: Getter y setter con `@property` y `@setter`.

Ejercicio 17 – Sobrecarga de operadores

Enunciado: Sobrecarga el operador + para sumar objetos de tipo `Vector`.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, otro):
        return Vector(self.x + otro.x, self.y + otro.y)

v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2
print(v3.x, v3.y)
```

Explicación:

- `__add__` permite **sobrecargar** el operador +.
- `v1 + v2` crea un nuevo objeto sumando atributos.

Comandos de POO explicados: Métodos especiales, sobrecarga de operadores, creación de nuevos objetos.

Ejercicio 18 – Encapsulamiento con validación

Enunciado: Modifica un atributo privado solo si cumple una condición.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.__edad = edad

    def set_edad(self, edad):
        if 0 < edad < 120:
            self.__edad = edad

    def get_edad(self):
        return self.__edad

p = Persona("Ana", 25)
p.set_edad(130)
print(p.get_edad())  # No cambia
```

Explicación:

- Control de acceso con `set_edad`.
- Se valida que la edad esté en un rango lógico.

Comandos de POO explicados: Encapsulamiento, validación en setters.

Ejercicio 19 – Herencia y super

Enunciado: Usa `super()` para extender un método de la clase base.

```
class Persona:
    def saludar(self):
        print("Hola")

class Estudiante(Persona):
    def saludar(self):
        super().saludar()
        print("Soy estudiante")

e = Estudiante()
e.saludar()
```

Explicación:

- `super().saludar()` llama al método original y luego añade funcionalidad.
- Combina comportamiento de clase base y derivada.

Comandos de POO explicados: Herencia, `super()`, extensión de métodos.

Ejercicio 20 – Composición avanzada

Enunciado: Un `Libro` tiene un `Autor`. Mostrar información completa.

```
class Autor:
    def __init__(self, nombre):
        self.nombre = nombre

class Libro:
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor

    def mostrar(self):
        print(f"{self.titulo} escrito por {self.autor.nombre}")

a = Autor("Gabriel García Márquez")
l = Libro("Cien Años de Soledad", a)
l.mostrar()
```

Explicación:

- `Libro` contiene un objeto `Autor`.
- Se accede a los atributos del objeto contenido para mostrar información.

Comandos de POO explicados: Composición, acceso a atributos de objetos anidados, métodos de clase.

Ejercicio 21 – Herencia múltiple con métodos iguales

Enunciado: Crea dos clases `A` y `B` con un método llamado `saludo`. Luego crea `C` que herede de ambas y observa cuál método se ejecuta.

```
class A:
    def saludo(self):
        print("Hola desde A")

class B:
    def saludo(self):
        print("Hola desde B")

class C(A, B):
    pass

c = C()
c.saludo()
```

Explicación:

- Python sigue el **MRO (Method Resolution Order)**, por lo que se ejecuta el método de la primera clase en la herencia (`A`).
- `C(A, B)` hereda de `A` primero, luego `B`.

Comandos de POO explicados: Herencia múltiple, MRO, resolución de métodos.

Ejercicio 22 – Método privado

Enunciado: Crea un método privado y un método público que lo llame.

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre

    def __metodo_privado(self):
        print(f"Hola desde privado {self.nombre}")

    def llamar_privado(self):
        self.__metodo_privado()

p = Persona("Ana")
p.llamar_privado()
```

Explicación:

- `__metodo_privado` es un método **privado**, accesible solo dentro de la clase.
- `llamar_privado` permite ejecutar el método privado desde fuera de la clase.

Comandos de POO explicados: Métodos privados, encapsulamiento.

Ejercicio 23 – Clase con contador de instancias

Enunciado: Cuenta cuántos objetos se crean de una clase.

```
class Persona:
    total = 0

    def __init__(self, nombre):
        self.nombre = nombre
        Persona.total += 1

p1 = Persona("Ana")
p2 = Persona("Luis")
print(Persona.total)
```

Explicación:

- `total` es un **atributo de clase** compartido por todos los objetos.
- Cada vez que se crea un objeto, se incrementa el contador.

Comandos de POO explicados: Atributos de clase, incremento en constructor.

Ejercicio 24 – Herencia y método extendido

Enunciado: Redefine un método y añade información adicional.

```
class Persona:
    def presentarse(self):
        print("Hola")

class Estudiante(Persona):
    def presentarse(self):
        print("Hola, soy estudiante")

e = Estudiante()
e.presentarse()
```

Explicación:

- Se redefine el método `presentarse` en la clase hija.
- Esto es **polimorfismo**.

Comandos de POO explicados: Herencia, redefinición de métodos, polimorfismo.

Ejercicio 25 – Método estático con validación

Enunciado: Crea un método estático que valide si un número es par.

```
class Utilidades:
    @staticmethod
    def es_par(n):
        return n % 2 == 0

print(Utilidades.es_par(10))
print(Utilidades.es_par(7))
```

Explicación:

- `@staticmethod` permite usar el método sin instanciar la clase.
- Retorna `True` si el número es par.

Comandos de POO explicados: Métodos estáticos, operadores aritméticos, retorno de valores.

Ejercicio 26 – Sobrecarga de operadores con strings

Enunciado: Sobrecarga + para concatenar nombres de dos objetos `Persona`.

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre

    def __add__(self, otra):
        return Persona(self.nombre + " " + otra.nombre)

p1 = Persona("Ana")
p2 = Persona("Luis")
p3 = p1 + p2
print(p3.nombre)
```

Explicación:

- `__add__` redefine + para crear un nuevo objeto con nombres concatenados.

Comandos de POO explicados: Sobrecarga de operadores, creación de objetos en operadores.

Ejercicio 27 – Clase con lista de objetos y métodos de filtrado

Enunciado: Crear una clase `Curso` que almacene estudiantes y devuelva los mayores de 18 años.

```
class Estudiante:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

class Curso:
    def __init__(self):
        self.alumnos = []

    def agregar(self, estudiante):
        self.alumnos.append(estudiante)

    def mayores_de_edad(self):
        return [e.nombre for e in self.alumnos if e.edad >= 18]

curso = Curso()
curso.agregar(Estudiante("Ana", 17))
curso.agregar(Estudiante("Luis", 20))
print(curso.mayores_de_edad())
```

Explicación:

- La lista `self.alumnos` contiene objetos.
- Se filtra usando comprensión de listas.

Comandos de POO explicados: Composición, listas de objetos, comprensión de listas.

Ejercicio 28 – Clase con método de clase y atributos de clase

Enunciado: Crear un método que devuelva cuántos objetos se han creado de la clase `Estudiante`.

```
class Estudiante:
    total = 0

    def __init__(self, nombre):
        self.nombre = nombre
        Estudiante.total += 1

    @classmethod
    def contar(cls):
        return cls.total

e1 = Estudiante("Ana")
e2 = Estudiante("Luis")
print(Estudiante.contar())
```

Explicación:

- `@classmethod` accede a la **clase**, no al objeto.
- `cls.total` devuelve el total de objetos creados.

Comandos de POO explicados: Métodos de clase, atributos de clase, contador de instancias.

Ejercicio 29 – Polimorfismo con lista de objetos

Enunciado: Lista de animales y método `hacer_sonido` que varía según la clase.

```
class Animal:
    def hacer_sonido(self):
        pass

class Perro(Animal):
    def hacer_sonido(self):
        print("Guau")

class Gato(Animal):
    def hacer_sonido(self):
        print("Miau")

animales = [Perro(), Gato()]
for a in animales:
    a.hacer_sonido()
```

Explicación:

- Cada objeto responde a `hacer_sonido` según su clase.
- Esto es **polimorfismo** en acción.

Comandos de POO explicados: Polimorfismo, listas de objetos, herencia.

Ejercicio 30 – Decorador de clase

Enunciado: Crear un decorador para mostrar información antes de ejecutar un método.

```
def decorador(func):
    def wrapper(*args, **kwargs):
        print("Ejecutando método...")
        return func(*args, **kwargs)
    return wrapper

class Persona:
    def __init__(self, nombre):
        self.nombre = nombre

    @decorador
    def saludar(self):
        print(f"Hola, soy {self.nombre}")

p = Persona("Ana")
p.saludar()
```

Explicación:

- `@decorador` envuelve el método `saludar`.
- Ejecuta código antes y después de la función original.

Comandos de POO explicados: Decoradores de métodos, funciones internas, `*args` y `**kwargs`.

Ejercicio 31 – Manejo de excepciones en métodos

Enunciado: Crea una clase `CuentaBancaria` con un método para retirar dinero y maneja la excepción si no hay suficiente saldo.

```
class CuentaBancaria:
    def __init__(self, titular, saldo):
        self.titular = titular
        self.saldo = saldo

    def retirar(self, monto):
        if monto > self.saldo:
            raise ValueError("Saldo insuficiente")
        self.saldo -= monto
        print(f"Retirado {monto}, saldo restante {self.saldo}")

cuenta = CuentaBancaria("Ana", 100)
cuenta.retirar(50)
# cuenta.retirar(100)  # Esto lanzaría una excepción
```

Explicación:

- `raise ValueError` lanza una **excepción** si la condición no se cumple.
- Se controla el flujo de error desde dentro del método.

Comandos de POO explicados: Excepciones, métodos, atributos, control de errores.

Ejercicio 32 – Herencia y excepciones

Enunciado: Clase base `CuentaBancaria` y subclase `CuentaAhorros` que lanza excepción si el retiro supera un límite.

```
class CuentaBancaria:
    def __init__(self, saldo):
        self.saldo = saldo

    def retirar(self, monto):
        if monto > self.saldo:
            raise ValueError("Saldo insuficiente")
        self.saldo -= monto

class CuentaAhorros(CuentaBancaria):
    def retirar(self, monto):
        if monto > 500:
            raise ValueError("Límite de retiro excedido")
        super().retirar(monto)

c = CuentaAhorros(1000)
c.retirar(400)
print(c.saldo)
```

Explicación:

- `super().retirar(monto)` llama al método de la clase base.
- Se añade una validación adicional específica de la subclase.

Comandos de POO explicados: Herencia, sobrescritura de métodos, excepciones.

Ejercicio 33 – Mixins

Enunciado: Crea un `Mixin` que añada la funcionalidad de imprimir información.

```
class InfoMixin:
    def mostrar_info(self):
        print(f"{self.__class__.__name__}: {self.__dict__}")

class Persona(InfoMixin):
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

p = Persona("Ana", 25)
p.mostrar_info()
```

Explicación:

- `Mixin` es una clase que proporciona **funcionalidad adicional** sin ser base principal.
- `self.__dict__` devuelve los atributos del objeto.

Comandos de POO explicados: Mixins, herencia, atributos de objetos.

Ejercicio 34 – Herencia múltiple con mixins

Enunciado: Combina `Mixin` con otra clase base.

```
class InfoMixin:
    def mostrar_info(self):
        print(f"{self.__class__.__name__}: {self.__dict__}")

class Estudiante:
    def __init__(self, nombre, curso):
        self.nombre = nombre
        self.curso = curso

class EstudianteInfo(Estudiante, InfoMixin):
    pass

e = EstudianteInfo("Luis", "Matemáticas")
e.mostrar_info()
```

Explicación:

- `EstudianteInfo` hereda de `Estudiante` y `InfoMixin`.
- Combina atributos de `Estudiante` con método de `InfoMixin`.

Comandos de POO explicados: Herencia múltiple, mixins, combinación de métodos.

Ejercicio 35 – Encapsulamiento avanzado con propiedades

Enunciado: Atributo privado con getter, setter y validación avanzada.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self._edad = edad

    @property
    def edad(self):
        return self._edad

    @edad.setter
    def edad(self, valor):
        if 0 < valor < 120:
            self._edad = valor
        else:
            print("Edad inválida")

p = Persona("Ana", 25)
p.edad = 130  # Edad inválida
print(p.edad)
```

Explicación:

- `@property` y `@setter` permiten controlar el acceso a un atributo privado.
- Se valida que los valores sean lógicos antes de modificarlos.

Comandos de POO explicados: Propiedades, encapsulamiento, validación de datos.

Ejercicio 36 – Sobrecarga de operadores con objetos complejos

Enunciado: Sobrecarga `*` para multiplicar vectores por un escalar.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __mul__(self, escalar):
        return Vector(self.x * escalar, self.y * escalar)

v = Vector(2, 3)
v2 = v * 5
print(v2.x, v2.y)
```

Explicación:

- `__mul__` redefine `*` para multiplicar atributos del objeto.
- Retorna un **nuevo objeto**.

Comandos de POO explicados: Sobrecarga de operadores, objetos inmutables.

Ejercicio 37 – Polimorfismo con múltiples subclases

Enunciado: Diferentes animales responden a `hacer_sonido` de manera distinta.

```
class Animal:
    def hacer_sonido(self):
        pass

class Perro(Animal):
    def hacer_sonido(self):
        print("Guau")

class Gato(Animal):
    def hacer_sonido(self):
        print("Miau")

class Vaca(Animal):
    def hacer_sonido(self):
        print("Muu")

animales = [Perro(), Gato(), Vaca()]
for a in animales:
    a.hacer_sonido()
```

Explicación:

- Cada subclase redefine `hacer_sonido`.
- Lista de objetos permite polimorfismo dinámico.

Comandos de POO explicados: Polimorfismo, herencia, listas de objetos.

Ejercicio 38 – Clase abstracta con múltiples métodos

Enunciado: Clase abstracta `Figura` obliga a implementar `area` y `perimetro`.

```
from abc import ABC, abstractmethod

class Figura(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimetro(self):
        pass

class Rectangulo(Figura):
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    def area(self):
        return self.base * self.altura

    def perimetro(self):
        return 2 * (self.base + self.altura)

r = Rectangulo(5, 3)
print(r.area(), r.perimetro())
```

Explicación:

- Clase abstracta define métodos obligatorios.
- Subclase implementa todos los métodos.

Comandos de POO explicados: Clases abstractas, métodos abstractos, herencia obligatoria.

Ejercicio 39 – Métodos mágicos `__eq__` y `__repr__`

Enunciado: Compara objetos y devuelve representación legible.

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre

    def __eq__(self, otra):
        return self.nombre == otra.nombre

    def __repr__(self):
        return f"Persona({self.nombre})"

p1 = Persona("Ana")
p2 = Persona("Ana")
print(p1 == p2)
print(p1)
```

Explicación:

- `__eq__` redefine `==` para comparar objetos.
- `__repr__` devuelve una representación legible del objeto.

Comandos de POO explicados: Métodos mágicos, sobrecarga de operadores, representación de objetos.

Ejercicio 40 – Clase con métodos de clase y estáticos

Enunciado: Clase `Temperatura` que convierte grados y mantiene un contador de conversiones.

```
class Temperatura:
    conversiones = 0

    @classmethod
    def contar_conversiones(cls):
        return cls.conversiones

    @staticmethod
    def celsius_a_fahrenheit(c):
        Temperatura.conversiones += 1
        return (c * 9/5) + 32

print(Temperatura.celsius_a_fahrenheit(0))
print(Temperatura.celsius_a_fahrenheit(100))
print(Temperatura.contar_conversiones())
```

Explicación:

- `@staticmethod` convierte grados sin usar instancia.
- `@classmethod` accede a contador de conversiones.
- Atributo de clase mantiene el estado compartido.

Comandos de POO explicados: Métodos estáticos y de clase, atributos de clase, contador global de objetos.

Ejercicio 41 – Patrón Singleton

Enunciado: Crear una clase `BaseDatos` que solo permita una instancia.

```
class BaseDatos:
    __instancia = None

    def __new__(cls, *args, **kwargs):
        if cls.__instancia is None:
            cls.__instancia = super().__new__(cls)
        return cls.__instancia

bd1 = BaseDatos()
bd2 = BaseDatos()
print(bd1 is bd2)  # True, misma instancia
```

Explicación:

- `__new__` controla la creación de objetos.
- Se asegura que solo exista una instancia (**Singleton**).

Comandos de POO explicados: `__new__`, atributos privados de clase, control de instanciación.

Ejercicio 42 – Factory Method

Enunciado: Crea un método que devuelva diferentes objetos según parámetro.

```
class Animal:
    def hacer_sonido(self):
        pass

class Perro(Animal):
    def hacer_sonido(self):
        print("Guau")

class Gato(Animal):
    def hacer_sonido(self):
        print("Miau")

def crear_animal(tipo):
    if tipo == "perro":
        return Perro()
    elif tipo == "gato":
        return Gato()

a1 = crear_animal("perro")
a2 = crear_animal("gato")
a1.hacer_sonido()
a2.hacer_sonido()
```

Explicación:

- `crear_animal` decide qué subclase instanciar (**Factory Method**).
- Retorna objeto según parámetro.

Comandos de POO explicados: Factory pattern, herencia, polimorfismo.

Ejercicio 43 – Composición profunda

Enunciado: Una clase `Empresa` que tiene varios `Departamento`, y cada departamento varios `Empleado`.

```
class Empleado:
    def __init__(self, nombre):
        self.nombre = nombre

class Departamento:
    def __init__(self, nombre):
        self.nombre = nombre
        self.empleados = []

    def agregar_empleado(self, empleado):
        self.empleados.append(empleado)

class Empresa:
    def __init__(self, nombre):
        self.nombre = nombre
        self.departamentos = []

    def agregar_departamento(self, depto):
        self.departamentos.append(depto)

d = Departamento("IT")
d.agregar_empleado(Empleado("Ana"))
d.agregar_empleado(Empleado("Luis"))
e = Empresa("TechCorp")
e.agregar_departamento(d)
for depto in e.departamentos:
    for emp in depto.empleados:
        print(emp.nombre)
```

Explicación:

- Se usan **objetos dentro de objetos** (composición profunda).
- Acceso a atributos anidados mediante bucles.

Comandos de POO explicados: Composición, listas de objetos, bucles anidados.

Ejercicio 44 – Mixin avanzado

Enunciado: Crear un mixin que añada funcionalidad de impresión JSON a cualquier clase.

```
import json

class JsonMixin:
    def to_json(self):
        return json.dumps(self.__dict__)

class Persona(JsonMixin):
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

p = Persona("Ana", 25)
print(p.to_json())
```

Explicación:

- `JsonMixin` añade un método a cualquier clase que herede de él.
- Se accede a atributos mediante `__dict__`.

Comandos de POO explicados: Mixins, herencia, serialización de objetos.

Ejercicio 45 – Propiedades con validación compleja

Enunciado: Atributo privado con getter, setter y validación de múltiples condiciones.

```
class Producto:
    def __init__(self, nombre, precio):
        self.nombre = nombre
        self._precio = precio

    @property
    def precio(self):
        return self._precio

    @precio.setter
    def precio(self, valor):
        if valor > 0:
            self._precio = valor
        else:
            print("Precio inválido")

p = Producto("Laptop", 1000)
p.precio = -50  # inválido
print(p.precio)
```

Explicación:

- Getter y setter permiten **controlar los datos** antes de asignarlos.
- Se evita que el precio sea negativo.

Comandos de POO explicados: Propiedades, validación, encapsulamiento.

Ejercicio 46 – Sobrecarga de operadores con comparación

Enunciado: Sobrecarga > para comparar objetos según un atributo.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __gt__(self, otra):
        return self.edad > otra.edad

p1 = Persona("Ana", 25)
p2 = Persona("Luis", 30)
print(p1 > p2)
```

Explicación:

- `__gt__` redefine > comparando atributos de objetos.
- Retorna `True` o `False`.

Comandos de POO explicados: Sobrecarga de operadores, comparación de objetos.

Ejercicio 47 – Herencia compleja

Enunciado: Clase base `Vehiculo`, subclases `Coche` y `Moto` con métodos propios.

```
class Vehiculo:
    def arrancar(self):
        print("Vehículo arrancado")

class Coche(Vehiculo):
    def abrir_maletero(self):
        print("Maletero abierto")

class Moto(Vehiculo):
    def hacer_caballito(self):
        print("Caballito realizado")

c = Coche()
m = Moto()
c.arrancar()
c.abrir_maletero()
m.arrancar()
m.hacer_caballito()
```

Explicación:

- Herencia simple con métodos específicos por subclase.
- Cada objeto tiene comportamiento propio y heredado.

Comandos de POO explicados: Herencia, métodos propios y heredados.

Ejercicio 48 – Polimorfismo con listas heterogéneas

Enunciado: Lista de distintos vehículos ejecutando `arrancar`.

```
class Vehiculo:
    def arrancar(self):
        pass

class Coche(Vehiculo):
    def arrancar(self):
        print("Coche arrancado")

class Moto(Vehiculo):
    def arrancar(self):
        print("Moto arrancada")

vehiculos = [Coche(), Moto(), Coche()]
for v in vehiculos:
    v.arrancar()
```

Explicación:

- Cada subclase redefine `arrancar`.
- Polimorfismo permite tratar todos los objetos como `Vehiculo`.

Comandos de POO explicados: Polimorfismo, listas de objetos, herencia.

Ejercicio 49 – Métodos mágicos combinados

Enunciado: Sobrecarga de `+` y `__str__` para objetos de tipo `Dinero`.

```
class Dinero:
    def __init__(self, cantidad):
        self.cantidad = cantidad

    def __add__(self, otra):
        return Dinero(self.cantidad + otra.cantidad)

    def __str__(self):
        return f"${self.cantidad}"

d1 = Dinero(50)
d2 = Dinero(75)
d3 = d1 + d2
print(d3)
```

Explicación:

- `__add__` permite sumar objetos `Dinero`.
- `__str__` permite mostrar el objeto de manera legible.

Comandos de POO explicados: Métodos mágicos, sobrecarga de operadores, representación de objetos.

Ejercicio 50 – Composición + mixins + propiedades

Enunciado: Una clase `Pedido` que contiene `Producto` y añade funcionalidad de total y JSON.

```
import json

class Producto:
    def __init__(self, nombre, precio):
        self.nombre = nombre
        self.precio = precio

class JsonMixin:
    def to_json(self):
        return json.dumps(self.__dict__)

class Pedido(JsonMixin):
    def __init__(self):
        self.productos = []

    @property
    def total(self):
        return sum(p.precio for p in self.productos)

    def agregar_producto(self, producto):
        self.productos.append(producto)

pedido = Pedido()
pedido.agregar_producto(Producto("Laptop", 1000))
pedido.agregar_producto(Producto("Mouse", 50))
print(pedido.total)
print(pedido.to_json())
```

Explicación:

- `Pedido` contiene varios objetos `Producto` (composición).
- `total` es una propiedad que calcula el valor total.
- `JsonMixin` añade funcionalidad de exportación a JSON.

Comandos de POO explicados: Composición profunda, mixins, propiedades, métodos de clase, listas de objetos.

MANUAL DETALLADO DE EJERCICIOS DE TRATAMIENTO DE FICHEROS EN PYTHON

BLOQUE 1: Lectura, Escritura y Modos Básicos de Ficheros de Texto

Ejercicio 1 – Crear y escribir un fichero de texto

- **Enunciado:** Crear un archivo llamado `archivo1.txt` y escribir la frase "Hola mundo".
- **Explicación Detallada:** Para realizar esta tarea, se utiliza la función `open()` especificando el nombre del archivo y el modo de apertura `"w"` (escritura). El uso de `with open(...)` garantiza que **el fichero se cierre automáticamente** después de que el bloque de código se complete. Finalmente, `f.write("Hola mundo")` inserta el contenido en el archivo.
- **Comandos y Controles Clave:**
    - `open()`: Función para abrir o crear un fichero.
    - Modo `w`: **Modo escritura**. Si el fichero existe, sobrescribe su contenido.
    - `with`: Asegura el **cierre automático** del fichero.
    - `write()`: Escribe la cadena de texto en el fichero.

Ejercicio 2 – Leer un fichero de texto

- **Enunciado:** Leer el contenido de `archivo1.txt` y mostrarlo en pantalla.
- **Explicación Detallada:** Se abre el fichero en modo lectura, indicado por `"r"`. La función `f.read()` lee **todo el contenido del fichero** y lo guarda en la variable `contenido`. Luego, `print(contenido)` lo muestra.
- **Comandos y Controles Clave:**
    - Modo `r`: **Modo lectura**.
    - `read()`: Lee todo el contenido del fichero.
    - Manejo de archivos con `with`.

Ejercicio 3 – Añadir texto a un fichero

- **Enunciado:** Añadir la línea "Segunda línea" a `archivo1.txt` **sin borrar el contenido anterior**.
- **Explicación Detallada:** Se utiliza el modo `"a"` (append) para abrir el fichero. Este modo está diseñado para añadir nuevo contenido al final del fichero sin sobrescribir el texto ya existente. Se incluye `\n` (salto de línea) antes del texto para asegurar que la nueva línea se escriba en una línea separada.
- **Comandos y Controles Clave:**
    - Modo `a`: **Modo añadir** (append).
    - `\n`: Añade **salto de línea**.
    - `write()`: Escribe el contenido.

Ejercicio 4 – Leer líneas por separado

- **Enunciado:** Leer `archivo1.txt` línea por línea y mostrarlas.
- **Explicación Detallada:** Se itera directamente sobre el objeto fichero (`for linea in f:`), lo cual permite leer el archivo **línea por línea** de manera eficiente. `linea.strip()`

se usa para **eliminar los saltos de línea** y cualquier espacio extra asociado a cada línea antes de imprimirla.

- **Comandos y Controles Clave:**
  - **Iteración sobre ficheros**: Lee línea a línea.
  - `strip()`: Elimina saltos de línea y espacios extra.

Ejercicio 5 – Copiar un fichero

- **Enunciado:** Crear una copia de `archivo1.txt` llamada `copia.txt`.
- **Explicación Detallada:** Este ejercicio implica dos pasos: primero, se lee todo el contenido del fichero de origen (`origen.read()`); segundo, se abre el fichero de destino (`copia.txt`) en modo escritura (`"w"`) y se escribe el contenido previamente leído (`destino.write(contenido)`).
- **Comandos y Controles Clave:**
  - Lectura (`read()`) y escritura (`write()`).
  - Manejo de **múltiples ficheros** mediante dos bloques `with open()`.

Ejercicio 6 – Contar líneas de un fichero

- **Enunciado:** Contar cuántas líneas tiene `archivo1.txt`.
- **Explicación Detallada:** Se utiliza el método `readlines()`, que lee el fichero completo y devuelve una **lista donde cada elemento es una línea**. Luego, la función `len(lineas)` se aplica a esta lista para obtener el número total de líneas.
- **Comandos y Controles Clave:**
  - `readlines()`: Devuelve una **lista** con cada línea.
  - `len()`: Devuelve el número de elementos (líneas) en la lista.

Ejercicio 7 – Buscar una palabra en un fichero

- **Enunciado:** Comprobar si la palabra "Hola" existe en `archivo1.txt`.
- **Explicación Detallada:** Se lee primero todo el contenido del fichero (`f.read()`). Luego se utiliza el operador `in` dentro de una estructura condicional (`if "Hola" in contenido`) para verificar si la subcadena está presente en el texto.
- **Comandos y Controles Clave:**
  - `read()`: Lectura completa del contenido.
  - Operador `in`: Comprueba la presencia de una subcadena.
  - **Condicionales** (`if/else`).

Ejercicio 8 – Contar palabras en un fichero

- **Enunciado:** Contar cuántas palabras tiene `archivo1.txt`.
- **Explicación Detallada:** Tras leer todo el contenido del fichero, se aplica el método `split()` a la cadena de texto, el cual **separa el texto por espacios** y devuelve una lista de palabras. Finalmente, se utiliza `len(palabras)` para obtener el número total de elementos (palabras) en la lista.
- **Comandos y Controles Clave:**
  - `split()`: Separa el texto en una lista de palabras.
  - `len()`: Cuenta el número total de palabras.

Ejercicio 9 – Escribir varias líneas desde una lista

- **Enunciado:** Crear `archivo2.txt` y escribir varias líneas desde una lista.
- **Explicación Detallada:** Se define una lista de cadenas. Se itera sobre esta lista (`for linea in lineas:`) y para cada elemento, se usa `f.write()` para escribir la línea en el fichero, asegurándose de **concatenar `\n`** al final de cada línea para el salto de línea.
- **Comandos y Controles Clave:**
  - Iteración sobre listas.
  - `write()`: Escritura de contenido.
  - Concatenación de strings (usando `+ "\n"`).

Ejercicio 10 – Leer fichero y convertir a lista

- **Enunciado:** Leer `archivo2.txt` y guardarlo en una lista de líneas **sin saltos de línea**.
- **Explicación Detallada:** Se emplea la **List comprehension** (comprensión de listas) para recorrer el fichero. Por cada línea leída (`for linea in f`), se aplica `linea.strip()` para eliminar los espacios y saltos de línea, creando así una lista limpia de contenidos.
- **Comandos y Controles Clave:**
  - **List comprehension**.
  - `strip()`: Limpia espacios y saltos de línea.
  - Lectura de ficheros mediante iteración.

BLOQUE 2: Manejo de Errores y Ficheros Estructurados (JSON, CSV, Binario)

Ejercicio 11 – Manejo de excepciones al abrir ficheros

- **Enunciado:** Abrir un fichero que puede no existir y mostrar un mensaje si ocurre un error.
- **Explicación Detallada:** Se utiliza el bloque `try` para intentar ejecutar la apertura y lectura del fichero. Si el fichero (`no_existe.txt`) no se encuentra, la excepción específica **`except FileNotFoundError:`** captura el error y evita que el programa se detenga, mostrando un mensaje amigable.
- **Comandos y Controles Clave:**
  - `try-except`: Estructura para manejar errores.
  - **`FileNotFoundError`**: Excepción específica capturada cuando un fichero no se encuentra.
  - `open()`, `read()`.

Ejercicio 12 – Escribir y leer ficheros JSON

- **Enunciado:** Guardar un diccionario en un fichero JSON y luego leerlo.
- **Explicación Detallada:** Se requiere importar el módulo `json`. Para escribir, `json.dump(data, f)` toma el objeto Python (`data`) y lo serializa (lo convierte a formato JSON) escribiéndolo en el fichero abierto. Para leer, `json.load(f)` lee el fichero JSON y lo deserializa, devolviendo un objeto Python (diccionario en este caso).
- **Comandos y Controles Clave:**
  - `import json`.
  - `json.dump()`: Escribe un objeto Python a un fichero JSON.
  - `json.load()`: Lee un fichero JSON y lo convierte a un objeto Python.

- Modos de apertura `w` y `r`.

Ejercicio 13 – Crear fichero CSV

- **Enunciado:** Crear un fichero CSV con información de personas.
- **Explicación Detallada:** Requiere `import csv`. `csv.writer(f)` crea un objeto que facilita la escritura en formato CSV. Se usa `writer.writerows(datos)` para **escribir múltiples filas** a partir de una lista de listas (`datos`). El parámetro `newline=""` en la función `open()` es crucial para evitar la inserción de líneas en blanco adicionales en el archivo CSV.
- **Comandos y Controles Clave:**
  - `import csv`.
  - `csv.writer`: Crea el objeto escritor de CSV.
  - `writerows()`: Escribe varias filas a la vez.
  - `newline=""`: Control para evitar líneas en blanco en el CSV.

Ejercicio 14 – Leer fichero CSV

- **Enunciado:** Leer `personas.csv` y mostrar sus contenidos.
- **Explicación Detallada:** Se usa `csv.reader(f)`, que crea un objeto iterable para leer el CSV fila por fila. Al iterar sobre este objeto (`for fila in reader:`), cada fila del CSV se devuelve como una **lista de strings**.
- **Comandos y Controles Clave:**
  - `csv.reader`: Permite leer el CSV fila por fila.
  - Iteración y manejo de ficheros.

Ejercicio 15 – Añadir a un CSV

- **Enunciado:** Añadir una nueva persona al CSV sin borrar el contenido.
- **Explicación Detallada:** Se abre el fichero en modo añadir (`"a"`). Se utiliza `writer.writerow(["Marta", 28])` para escribir una **sola fila** al final del archivo.
- **Comandos y Controles Clave:**
  - `writer.writerow()`: Escribe una única fila.
  - Modo `a`: Añade contenido.

Ejercicio 16 – Buscar una palabra en un CSV

- **Enunciado:** Ver si existe una persona llamada "Luis" en `personas.csv`.
- **Explicación Detallada:** Se itera a través de cada fila devuelta por `csv.reader`. Se usa el operador `in` para comprobar si el valor "Luis" está presente en la lista que representa la fila actual (`if "Luis" in fila:`). Si se encuentra, se imprime la fila completa.
- **Comandos y Controles Clave:**
  - Bucle `for` sobre el objeto CSV reader.
  - Operador `in` para búsqueda en lista.

Ejercicio 17 – Contar líneas de un CSV

- **Enunciado:** Contar cuántas personas hay (excluyendo la cabecera).

- **Explicación Detallada:** Para ignorar la fila de cabecera, se utiliza la función **`next(reader)`**, la cual avanza el iterador una posición. Luego, se usa `list(reader)` para convertir las filas restantes (solo los datos) en una lista, y `len(filas)` para obtener el conteo de registros.
- **Comandos y Controles Clave:**
    - `next()`: Salta la primera fila (cabecera).
    - `list()`: Convierte el iterador restante a una lista.
    - `len()`: Cuenta el número de personas.

Ejercicio 18 – Fichero binario

- **Enunciado:** Guardar una lista de números en binario y leerla.
- **Explicación Detallada:** Requiere `import pickle`. Para escribir en binario, se usa el modo **`wb`** (write binary) y `pickle.dump(objeto, fichero)` para serializar y guardar el objeto Python en ese formato. Para leer, se usa el modo **`rb`** (read binary) y `pickle.load(fichero)` para deserializar y recuperar el objeto Python original.
- **Comandos y Controles Clave:**
    - `import pickle`.
    - Modos `wb/rb`: Indica manejo binario.
    - `dump()`: Guarda un objeto en formato binario.
    - `load()`: Lee un objeto desde binario.

BLOQUE 3: Eficiencia, Codificación y Manipulación de Contenido

Ejercicio 19 – Leer ficheros grandes línea por línea

- **Enunciado:** Leer un fichero grande **sin cargar todo en memoria**.
- **Explicación Detallada:** La lectura eficiente se logra simplemente iterando sobre el objeto fichero (`for linea in f:`). Este proceso lee el contenido línea por línea, lo cual es ideal para ficheros grandes, ya que solo una línea a la vez se mantiene en la memoria.
- **Comandos y Controles Clave:**
    - **Iteración sobre ficheros**: Lectura eficiente.
    - `strip()`: Limpieza de líneas.

Ejercicio 20 – Copiar fichero binario

- **Enunciado:** Copiar una imagen de un fichero binario.
- **Explicación Detallada:** Para asegurar que se copien correctamente datos no textuales (como imágenes o audio), se deben usar los modos binarios: **`rb`** para leer el origen y **`wb`** para escribir el destino. Se lee el contenido completo con `origen.read()` y se escribe en el destino con `destino.write(datos)`.
- **Comandos y Controles Clave:**
    - Modos binarios `rb/wb`.
    - `read()`, `write()`.

Ejercicio 21 – Leer ficheros con encoding específico

- **Enunciado:** Abrir un fichero `archivo_utf8.txt` con codificación UTF-8 y leer su contenido.
- **Explicación Detallada:** Al abrir el fichero, se añade el parámetro **`encoding="utf-8"`** a la función `open()`. Esto garantiza que los caracteres especiales o acentuados sean interpretados correctamente al leer el contenido con `read()`.
- **Comandos y Controles Clave:**
    - Parámetro `encoding`: Especifica el esquema de codificación.
    - `read()`, `open()`, `with`.

Ejercicio 22 – Escribir ficheros con encoding específico

- **Enunciado:** Crear `archivo_utf8.txt` y escribir texto con caracteres especiales.
- **Explicación Detallada:** Similar a la lectura, el parámetro `encoding="utf-8"` debe usarse al abrir el fichero en modo escritura (`"w"`). Esto permite que se escriban y almacenen correctamente caracteres acentuados y símbolos.
- **Comandos y Controles Clave:**
    - `write()`.
    - Modo `w`.
    - Codificación `UTF-8`.

Ejercicio 23 – Crear y usar ficheros temporales

- **Enunciado:** Crear un fichero temporal, escribir y leer su contenido.
- **Explicación Detallada:** Requiere `import tempfile`. **`tempfile.TemporaryFile("w+")`** crea un archivo temporal que tiene la propiedad de **eliminarse automáticamente** cuando se cierra. El modo `"w+"` permite tanto escribir (`w`) como leer (`+`). Después de escribir, se debe usar **`f.seek(0)`** para mover el puntero de lectura de nuevo al inicio del archivo antes de poder leer su contenido.
- **Comandos y Controles Clave:**
    - `import tempfile`, `TemporaryFile()`.
    - `seek(0)`: Mueve el puntero al inicio del fichero.
    - Modo `"w+"`: Escritura y lectura.

Ejercicio 24 – Buscar y reemplazar texto en un fichero

- **Enunciado:** Reemplazar "Hola" por "Adiós" en `archivo1.txt`.
- **Explicación Detallada:** Se realiza leyendo el contenido completo, modificando la cadena de texto en memoria usando el método **`contenido.replace("Hola", "Adiós")`**. Luego, el fichero se abre de nuevo en modo escritura (`"w"`) para **sobrescribir**lo con el contenido modificado.
- **Comandos y Controles Clave:**
    - `replace()`: Sustituye subcadenas en un *string*.
    - Lectura y reescritura de ficheros.

Ejercicio 25 – Registrar logs en un fichero

- **Enunciado:** Crear un fichero de logs y escribir mensajes con marca de tiempo.

- **Explicación Detallada:** Requiere importar `datetime`. Se abre el fichero `log.txt` en modo añadir (`"a"`). Se usa `datetime.now()` para obtener la fecha y hora actuales, y se inserta este valor junto al mensaje usando **f-strings**. Este método es útil para registrar eventos o errores, añadiéndolos secuencialmente.
- **Comandos y Controles Clave:**
    - `import datetime`, `datetime.now()`.
    - Modo `a`: Añadir la línea al final.
    - f-strings (cadenas formateadas).

Ejercicio 26 – Leer ficheros de logs línea por línea

- **Enunciado:** Mostrar solo las líneas que contienen la palabra "error" en `log.txt`.
- **Explicación Detallada:** Se itera sobre el fichero línea por línea. Para realizar una búsqueda que **no distinga entre mayúsculas y minúsculas**, se convierte la línea a minúsculas usando `linea.lower()` antes de buscar la subcadena `"error"`.
- **Comandos y Controles Clave:**
    - Iteración sobre ficheros.
    - `lower()`: Convierte la cadena a minúsculas.
    - Condicionales.

BLOQUE 4: Procesamiento Avanzado de Ficheros y Rutas

Ejercicio 27 – Dividir un fichero en varios ficheros pequeños

- **Enunciado:** Dividir `archivo1.txt` en ficheros de 2 líneas cada uno.
- **Explicación Detallada:** Primero, `readlines()` obtiene todas las líneas en una lista. Se usa un bucle `for` con *slicing* de listas (`lineas[i:i+2]`) y un paso de 2 (`range(0, len(lineas), 2)`) para seleccionar bloques de 2 líneas. **`f_parte.writelines()`** escribe las líneas seleccionadas en el nuevo fichero de forma eficiente.
- **Comandos y Controles Clave:**
    - `readlines()`, `writelines()`.
    - Bucles `for` con paso.
    - *Slicing* de listas.

Ejercicio 28 – Combinar varios ficheros en uno

- **Enunciado:** Combinar `parte_1.txt` y `parte_2.txt` en `combinado.txt`.
- **Explicación Detallada:** Se itera sobre una lista de nombres de ficheros. Para cada archivo, se lee su contenido completo (`f.read()`) y se escribe en el único fichero de destino (`destino.write()`). Esto fusiona los datos secuencialmente.
- **Comandos y Controles Clave:**
    - Iteración, `open()`, `read()`, `write()`.

Ejercicio 29 – Leer CSV con encabezado usando DictReader

- **Enunciado:** Leer `personas.csv` y acceder a cada campo por nombre.
- **Explicación Detallada:** Se utiliza **`csv.DictReader(f)`** en lugar de `csv.reader`. `DictReader` usa la primera fila como cabecera y devuelve cada fila subsiguiente como un

diccionario, permitiendo el acceso a los datos por el nombre de la columna (ej: `fila["Nombre"]`).

- **Comandos y Controles Clave:**
  - `csv.DictReader`.
  - Acceso a diccionarios por clave.

Ejercicio 30 – Escribir CSV con DictWriter

- **Enunciado:** Crear un CSV usando diccionarios.
- **Explicación Detallada:** Se usa **`csv.DictWriter(f, fieldnames=campos)`**. Se debe especificar explícitamente la lista de nombres de campo (`campos`) para que el escritor sepa el orden. **`writer.writeheader()`** escribe la fila de cabecera automáticamente. Finalmente, `writer.writerows(datos)` escribe los diccionarios como filas.
- **Comandos y Controles Clave:**
  - `csv.DictWriter`.
  - `writeheader()`: Escribe la cabecera.
  - `writerows()`: Escribe múltiples filas desde una lista de diccionarios.

Ejercicio 31 – Leer y escribir ficheros usando rutas completas

- **Enunciado:** Crear un fichero en una carpeta específica y leerlo desde su ruta absoluta.
- **Explicación Detallada:** Requiere `import os`. **`os.path.join("carpeta", "archivo_ruta.txt")`** construye la ruta de manera segura, adaptándose al sistema operativo. **`os.makedirs("carpeta", exist_ok=True)`** crea el directorio si no existe, y `exist_ok=True` previene errores si ya existe. Luego se usan rutas completas para `open()` y `read()/write()`.
- **Comandos y Controles Clave:**
  - `os.path.join()`: Construcción segura de rutas.
  - `os.makedirs(..., exist_ok=True)`: Creación de carpetas.
  - Modos `w` y `r` con rutas completas.

Ejercicio 32 – Renombrar un fichero

- **Enunciado:** Renombrar `archivo_ruta.txt` a `archivo_renombrado.txt`.
- **Explicación Detallada:** Se utiliza la función **`os.rename(ruta_antigua, ruta_nueva)`**. Esta función cambia el nombre de un fichero o carpeta. Si el archivo especificado no existe, se generará un error.
- **Comandos y Controles Clave:**
  - `os.rename()`.

Ejercicio 33 – Borrar un fichero

- **Enunciado:** Eliminar el fichero `archivo_renombrado.txt`.
- **Explicación Detallada:** Se usa **`os.remove(ruta_fichero)`**. Al igual que con `os.rename`, si el fichero no existe, se lanzará un `FileNotFoundError`.
- **Comandos y Controles Clave:**
  - `os.remove()`.

Ejercicio 34 – Crear y leer fichero CSV con delimitador personalizado

- **Enunciado:** Crear un CSV separado por `;` y leerlo.
- **Explicación Detallada:** Se pasa el parámetro **`delimiter=";"`** a `csv.writer()` al escribir, para que use el punto y coma en lugar de la coma. Este mismo parámetro debe ser usado en `csv.reader()` al leer el fichero para que interprete correctamente las separaciones.
- **Comandos y Controles Clave:**
  - Parámetro `delimiter` en `csv.writer` y `csv.reader`.
  - `writerows()`, `iteración`.

BLOQUE 5: Compresión, Utilidades de Sistema y Contadores

Ejercicio 35 – Comprimir un fichero con gzip

- **Enunciado:** Comprimir `archivo1.txt` en `archivo1.txt.gz`.
- **Explicación Detallada:** Requiere `import gzip`. Se abre el fichero original en modo binario de lectura (`"rb"`) y el fichero de destino se abre con **`gzip.open(..., "wb")`** (write binary, comprimido). Se utiliza `f_out.writelines(f_in)` para leer las líneas del original y escribirlas en el comprimido.
- **Comandos y Controles Clave:**
  - `gzip.open()`: Abre un fichero para compresión/descompresión gzip.
  - Modos binarios `rb` y `wb`.
  - `writelines()`.

Ejercicio 36 – Leer fichero gzip

- **Enunciado:** Leer el contenido de `archivo1.txt.gz`.
- **Explicación Detallada:** Se abre el archivo comprimido usando `gzip.open(..., "rb")`. `f.read()` devuelve el contenido como **bytes**. Para mostrarlo como texto legible, se debe usar **`.decode("utf-8")`**.
- **Comandos y Controles Clave:**
  - `gzip.open()`.
  - `decode("utf-8")`: Convierte bytes a string legible.
  - Modos binarios.

Ejercicio 37 – Copiar fichero usando shutil

- **Enunciado:** Copiar `archivo1.txt` a `copia_shutil.txt`.
- **Explicación Detallada:** Requiere `import shutil`. La función **`shutil.copy(origen, destino)`** realiza una copia simple del contenido del fichero, manteniendo los permisos básicos (aunque no los metadatos completos).
- **Comandos y Controles Clave:**
  - `import shutil`, `shutil.copy()`.

Ejercicio 38 – Mover fichero con shutil

- **Enunciado:** Mover `copia_shutil.txt` a la carpeta `carpeta`.

- **Explicación Detallada:** Se utiliza **`shutil.move(origen, destino)`**. Esta función mueve o renombra un fichero a una nueva ubicación. Si la carpeta de destino no existe, se puede usar `os.makedirs()` previamente para crearla.
- **Comandos y Controles Clave:**
  - `shutil.move()`.
  - `os.makedirs()`.

Ejercicio 39 – Contar caracteres, palabras y líneas

- **Enunciado:** Mostrar número de líneas, palabras y caracteres en `archivo1.txt`.
- **Explicación Detallada:**
  - Líneas: Se usa `readlines()` y `len()`.
  - Palabras: Se itera sobre cada línea y se aplica `linea.split()` (para obtener las palabras de esa línea), luego `len()` (para contar esas palabras), y finalmente `sum()` para acumular los conteos de todas las líneas.
  - Caracteres: Se itera sobre cada línea, se usa `len(linea)` para obtener la longitud, y se usa `sum()` para obtener el total.
- **Comandos y Controles Clave:**
  - `readlines()`, `split()`, `len()`, `sum()`, iteración.

Ejercicio 40 – Buscar múltiples palabras en un fichero

- **Enunciado:** Comprobar si "Hola" o "Mundo" aparecen en `archivo1.txt`.
- **Explicación Detallada:** Se define una lista de las palabras a buscar. Se lee todo el contenido del fichero. Luego, se **itera sobre la lista de palabras** y se usa el operador `in` para verificar si cada una está presente en el contenido total.
- **Comandos y Controles Clave:**
  - Iteración sobre listas.
  - Operador `in`, `read()`.

BLOQUE 6: JSON Avanzado y Manejo de Bloques Binarios

Ejercicio 41 – Leer JSON con lista de diccionarios

- **Enunciado:** Leer un fichero `datos.json` que contiene una lista de diccionarios y mostrar cada entrada.
- **Explicación Detallada:** `json.load(f)` convierte el contenido JSON directamente en una lista de objetos Python. Luego, se itera sobre esta lista y se accede a los campos de cada diccionario (ítem) por su clave (ej: `item['Nombre']`).
- **Comandos y Controles Clave:**
  - `json.load()`.
  - Iteración sobre listas de diccionarios.

Ejercicio 42 – Guardar lista de diccionarios en JSON

- **Enunciado:** Crear `datos.json` a partir de una lista de diccionarios.

- **Explicación Detallada:** `json.dump(lista, f)` guarda la lista. Se utiliza el parámetro **`indent=4`** para formatear el JSON con una indentación de 4 espacios, lo que lo hace mucho más legible para humanos.
- **Comandos y Controles Clave:**
  - `json.dump()`.
  - `indent`: Formatea la salida JSON.

Ejercicio 43 – Manejo de excepciones al leer JSON

- **Enunciado:** Leer JSON y controlar errores específicos (fichero inexistente y formato JSON incorrecto).
- **Explicación Detallada:** Se usan bloques `try-except` múltiples. Se captura `FileNotFoundError` si el archivo no está presente. Además, se captura **`json.JSONDecodeError`** si el archivo existe pero su contenido no está en un formato JSON válido (un error de sintaxis al parsear los datos).
- **Comandos y Controles Clave:**
  - `try-except`.
  - `json.JSONDecodeError`: Captura errores de formato JSON.
  - `FileNotFoundError`.

Ejercicio 44 – Leer y escribir ficheros grandes por bloques

- **Enunciado:** Copiar un fichero grande leyendo bloques de 1024 bytes.
- **Explicación Detallada:** Se abren ambos ficheros en modo binario (`rb` y `wb`). Se utiliza un bucle `while True`. **`origen.read(1024)`** lee solo 1024 bytes cada vez. El bucle se rompe (`break`) cuando `read()` devuelve un bloque vacío (`if not bloque:`), indicando el final del fichero. Esto es clave para evitar cargar ficheros muy grandes en la memoria.
- **Comandos y Controles Clave:**
  - `read(tamaño)`: Lectura por bloques.
  - Lectura binaria `rb`, escritura `wb`.
  - Bucle `while True` y control de flujo.

Ejercicio 45 – Añadir registro con timestamp en CSV

- **Enunciado:** Añadir filas a un CSV con fecha y hora actual.
- **Explicación Detallada:** Similar al Ejercicio 25, se utiliza `datetime.now()`. Se abre el CSV en modo añadir (`"a"`) y `writerow()` inserta una nueva fila que incluye la marca de tiempo junto con la descripción del evento.
- **Comandos y Controles Clave:**
  - `datetime.now()`, `csv.writer()`, `writerow()`.
  - Modo `a`.

Ejercicio 46 – Leer ficheros con múltiples tipos de salto de línea

- **Enunciado:** Leer un fichero independientemente del salto de línea usado (`\n`, `\r\n`).

- **Explicación Detallada:** Al abrir el fichero, se usa el parámetro **`newline=None`** en `open()`. Esto permite a Python **detectar y manejar automáticamente** los diferentes tipos de salto de línea utilizados en varios sistemas operativos.
- **Comandos y Controles Clave:**
  - `newline=None` en `open()`.
  - `strip()`.

Ejercicio 47 – Contar frecuencia de palabras

- **Enunciado:** Contar cuántas veces aparece cada palabra en un fichero.
- **Explicación Detallada:** Se utiliza la clase **`Counter`** del módulo `collections`. Después de leer el fichero y usar `split()` para obtener la lista de todas las palabras, `Counter(palabras)` calcula automáticamente la frecuencia de aparición de cada palabra.
- **Comandos y Controles Clave:**
  - `from collections import Counter`.
  - `Counter()`: Cuenta la frecuencia.
  - `split()`.

Ejercicio 48 – Ficheros binarios y struct

- **Enunciado:** Escribir números enteros en binario y leerlos.
- **Explicación Detallada:** Requiere `import struct`. Para escribir, se usa **`struct.pack("i", n)`**, donde `"i"` indica que se empaqueta un entero, convirtiendo el número en sus bytes correspondientes. Para leer, se usa `f.read(4)` para leer bloques de 4 bytes (el tamaño de un entero estándar). **`struct.unpack("i", bytes_leidos)`** convierte esos 4 bytes de vuelta a un entero.
- **Comandos y Controles Clave:**
  - `struct.pack()`: Convierte objetos Python a una representación binaria.
  - `struct.unpack()`: Convierte bytes a objetos Python.
  - Lectura y escritura binaria (`rb/wb`).

Ejercicio 49 – Filtrar líneas por longitud

- **Enunciado:** Crear un nuevo fichero solo con las líneas que tengan más de 10 caracteres.
- **Explicación Detallada:** Se abren dos ficheros simultáneamente (`f_in` para lectura y `f_out` para escritura). Se itera sobre el fichero de entrada. Dentro del bucle, la condición **`if len(linea.strip()) > 10:`** se usa para medir la longitud de la línea *sin* incluir saltos de línea ni espacios (`strip()`). Solo las líneas que cumplen la condición se escriben en el fichero de salida.
- **Comandos y Controles Clave:**
  - `len()`, `strip()`, `write()`.
  - Manejo de múltiples ficheros en un solo `with`.

Ejercicio 50 – Registrar errores en fichero log

- **Enunciado:** Capturar errores de ejecución y guardarlos en `errores.log`.

- **Explicación Detallada:** Se usa la estructura `try-except Exception as e:` para **capturar cualquier tipo de error** que ocurra durante la ejecución. El objeto error capturado (`e`) se escribe en el fichero `errores.log`, que se abre en modo añadir (`"a"`) para registrar todos los fallos secuencialmente sin sobrescribir el historial.
- **Comandos y Controles Clave:**
    - Manejo de excepciones (`try/except`).
    - `Exception as e`: Captura de errores generales.
    - Modo `a`.
    - `write()`.

Ejercicio 1 – Crear y escribir un fichero de texto

Enunciado: Crear un archivo llamado `archivo1.txt` y escribir la frase "Hola mundo".

```
with open("archivo1.txt", "w") as f:
    f.write("Hola mundo")
```

Explicación:

- `open("archivo1.txt", "w")` abre un fichero en **modo escritura** (`w`).
- `with` asegura que el fichero se cierre automáticamente.
- `f.write("Hola mundo")` escribe el contenido en el fichero.

Comandos explicados: `open()`, modos de apertura (`w`), `with`, `write()`, cierre automático.

Ejercicio 2 – Leer un fichero de texto

Enunciado: Leer el contenido de `archivo1.txt` y mostrarlo en pantalla.

```
with open("archivo1.txt", "r") as f:
    contenido = f.read()
    print(contenido)
```

Explicación:

- `"r"` abre el fichero en **modo lectura**.
- `f.read()` lee todo el contenido del fichero.
- `print()` muestra el contenido.

Comandos explicados: `read()`, modo lectura (`r`), manejo de archivos con `with`.

Ejercicio 3 – Añadir texto a un fichero

Enunciado: Añadir la línea "Segunda línea" a `archivo1.txt` sin borrar el contenido anterior.

```
with open("archivo1.txt", "a") as f:
    f.write("\nSegunda línea")
```

Explicación:

- `"a"` abre el fichero en **modo añadir**.
- `\n` añade salto de línea.
- No se borra el contenido anterior.

Comandos explicados: Modo `a` (append), salto de línea, `write()`.

Ejercicio 4 – Leer líneas por separado

Enunciado: Leer `archivo1.txt` línea por línea y mostrarlas.

```
with open("archivo1.txt", "r") as f:
    for linea in f:
        print(linea.strip())
```

Explicación:

- Iterar sobre el fichero devuelve **cada línea**.
- `strip()` elimina saltos de línea y espacios extra.

Comandos explicados: Iteración sobre ficheros, `strip()`, lectura línea a línea.

Ejercicio 5 – Copiar un fichero

Enunciado: Crear una copia de `archivo1.txt` llamada `copia.txt`.

```
with open("archivo1.txt", "r") as origen:
    contenido = origen.read()
with open("copia.txt", "w") as destino:
    destino.write(contenido)
```

Explicación:

- Se lee el contenido del fichero original y se escribe en otro fichero.
- `origen.read()` obtiene todo el contenido.

Comandos explicados: Lectura y escritura, `open()`, `write()`, manejo de múltiples ficheros.

Ejercicio 6 – Contar líneas de un fichero

Enunciado: Contar cuántas líneas tiene `archivo1.txt`.

```
with open("archivo1.txt", "r") as f:
    lineas = f.readlines()
    print(len(lineas))
```

Explicación:

- `readlines()` devuelve una lista con cada línea.
- `len(lineas)` devuelve el número de líneas.

Comandos explicados: `readlines()`, listas, `len()`.

Ejercicio 7 – Buscar una palabra en un fichero

Enunciado: Comprobar si la palabra "Hola" existe en `archivo1.txt`.

```
with open("archivo1.txt", "r") as f:
    contenido = f.read()
    if "Hola" in contenido:
        print("Palabra encontrada")
    else:
        print("No encontrada")
```

Explicación:

- Se lee todo el contenido.
- `in` comprueba si una subcadena está presente.

Comandos explicados: `read()`, operador `in`, condicionales.

Ejercicio 8 – Contar palabras en un fichero

Enunciado: Contar cuántas palabras tiene `archivo1.txt`.

```
with open("archivo1.txt", "r") as f:
    contenido = f.read()
    palabras = contenido.split()
    print(len(palabras))
```

Explicación:

- `split()` separa el texto por espacios.
- `len(palabras)` da el número total de palabras.

Comandos explicados: `split()`, listas, `len()`, lectura de ficheros.

Ejercicio 9 – Escribir varias líneas desde una lista

Enunciado: Crear `archivo2.txt` y escribir varias líneas desde una lista.

```
lineas = ["Primera", "Segunda", "Tercera"]
with open("archivo2.txt", "w") as f:
    for linea in lineas:
        f.write(linea + "\n")
```

Explicación:

- Se recorre la lista de líneas y se escribe cada una.
- `\n` añade salto de línea.

Comandos explicados: Iteración, `write()`, concatenación de strings.

Ejercicio 10 – Leer fichero y convertir a lista

Enunciado: Leer `archivo2.txt` y guardarlo en una lista de líneas sin saltos de línea.

```
with open("archivo2.txt", "r") as f:
    lineas = [linea.strip() for linea in f]
print(lineas)
```

Explicación:

- Se usa **list comprehension** para limpiar cada línea.
- `strip()` elimina espacios y saltos de línea.

Comandos explicados: List comprehension, `strip()`, lectura de ficheros.

Ejercicio 11 – Manejo de excepciones al abrir ficheros

Enunciado: Abrir un fichero que puede no existir y mostrar un mensaje si ocurre un error.

```
try:
    with open("no_existe.txt", "r") as f:
        contenido = f.read()
except FileNotFoundError:
    print("El fichero no existe")
```

Explicación:

- `try` intenta ejecutar el bloque de código.
- `except FileNotFoundError` captura la excepción si el fichero no existe.
- Evita que el programa se detenga.

Comandos explicados: `try-except`, `FileNotFoundError`, `open()`, `read()`.

Ejercicio 12 – Escribir y leer ficheros JSON

Enunciado: Guardar un diccionario en un fichero JSON y luego leerlo.

```
import json

data = {"nombre": "Ana", "edad": 25}

with open("data.json", "w") as f:
    json.dump(data, f)

with open("data.json", "r") as f:
    contenido = json.load(f)
    print(contenido)
```

Explicación:

- `json.dump()` escribe un objeto Python en un fichero JSON.
- `json.load()` lee el fichero y devuelve un objeto Python.

Comandos explicados: `import json`, `json.dump()`, `json.load()`, modos de apertura `w` y `r`.

Ejercicio 13 – Crear fichero CSV

Enunciado: Crear un fichero CSV con información de personas.

```
import csv

datos = [["Nombre", "Edad"], ["Ana", 25], ["Luis", 30]]

with open("personas.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerows(datos)
```

Explicación:

- `csv.writer(f)` crea un objeto para escribir CSV.
- `writerows()` escribe varias filas de la lista `datos`.

Comandos explicados: `import csv`, `csv.writer`, `writerows()`, `newline=""` para evitar líneas en blanco.

Ejercicio 14 – Leer fichero CSV

Enunciado: Leer `personas.csv` y mostrar sus contenidos.

```
import csv


with open("personas.csv", "r") as f:
    reader = csv.reader(f)
    for fila in reader:
        print(fila)
```

Explicación:

- `csv.reader(f)` permite leer CSV fila por fila.
- Iterar sobre `reader` devuelve cada fila como lista.

Comandos explicados: `csv.reader`, iteración, manejo de ficheros.

Ejercicio 15 – Añadir a un CSV

Enunciado: Añadir una nueva persona al CSV sin borrar el contenido.

```
import csv


with open("personas.csv", "a", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["Marta", 28])
```

Explicación:

- Modo `a` añade contenido al fichero.
- `writer.writerow()` escribe una sola fila.

Comandos explicados: `writer.writerow()`, modo `a`, CSV.

Ejercicio 16 – Buscar una palabra en un CSV

Enunciado: Ver si existe una persona llamada "Luis" en `personas.csv`.

```
import csv


with open("personas.csv", "r") as f:
    reader = csv.reader(f)
    for fila in reader:
        if "Luis" in fila:
            print("Encontrado:", fila)
```

Explicación:

- Se itera por filas y se usa `in` para buscar el valor.
- Se imprime la fila completa si coincide.

Comandos explicados: `for` sobre CSV, operador `in`, condicionales.

Ejercicio 17 – Contar líneas de un CSV

Enunciado: Contar cuántas personas hay (excluyendo cabecera).

```
import csv

with open("personas.csv", "r") as f:
    reader = csv.reader(f)
    next(reader)  # Saltar cabecera
    filas = list(reader)
    print("Número de personas:", len(filas))
```

Explicación:

- `next(reader)` salta la primera fila (cabecera).
- `list(reader)` convierte el resto en lista para contar.

Comandos explicados: `next()`, `list()`, `len()`, CSV.

Ejercicio 18 – Fichero binario

Enunciado: Guardar una lista de números en binario y leerla.

```
import pickle

numeros = [1, 2, 3, 4, 5]

with open("numeros.bin", "wb") as f:
    pickle.dump(numeros, f)

with open("numeros.bin", "rb") as f:
    datos = pickle.load(f)
    print(datos)
```

Explicación:

- `pickle.dump()` guarda objeto en formato binario.
- `pickle.load()` lee el objeto desde binario.

Comandos explicados: `import pickle`, `wb`/`rb` para binario, `dump()`, `load()`.

Ejercicio 19 – Leer ficheros grandes línea por línea

Enunciado: Leer un fichero grande sin cargar todo en memoria.

```
with open("archivo1.txt", "r") as f:
    for linea in f:
        print(linea.strip())
```

Explicación:

- Iterar sobre el fichero lee línea por línea.
- Evita cargar todo el fichero en memoria.

Comandos explicados: Iteración sobre ficheros, `strip()`, lectura eficiente.

Ejercicio 20 – Copiar fichero binario

Enunciado: Copiar una imagen de un fichero binario.

```
with open("imagen.jpg", "rb") as origen:
    datos = origen.read()
with open("copia_imagen.jpg", "wb") as destino:
    destino.write(datos)
```

Explicación:

- `rb` lee en **modo binario**.
- `wb` escribe en **modo binario**.
- Se puede copiar cualquier fichero binario (imágenes, audio, etc.).

Comandos explicados: Modos binarios `rb`/`wb`, `read()`, `write()`, copiado de ficheros.

Ejercicio 21 – Leer ficheros con encoding específico

Enunciado: Abrir un fichero `archivo_utf8.txt` con codificación UTF-8 y leer su contenido.

```
with open("archivo_utf8.txt", "r", encoding="utf-8") as f:
    contenido = f.read()
    print(contenido)
```

Explicación:

- `encoding="utf-8"` asegura que se interpreten correctamente caracteres especiales.
- `read()` lee todo el contenido del fichero.

Comandos explicados: `open()`, parámetro `encoding`, `read()`, `with`.

Ejercicio 22 – Escribir ficheros con encoding específico

Enunciado: Crear `archivo_utf8.txt` y escribir texto con caracteres especiales.

```
texto = "Hola, ¿cómo estás?"

with open("archivo_utf8.txt", "w", encoding="utf-8") as f:
    f.write(texto)
```

Explicación:

- Modo `"w"` escribe el contenido en el fichero.
- `encoding="utf-8"` permite usar caracteres acentuados y símbolos.

Comandos explicados: `write()`, modo `w`, codificación UTF-8.

Ejercicio 23 – Crear y usar ficheros temporales

Enunciado: Crear un fichero temporal, escribir y leer su contenido.

```
import tempfile


with tempfile.TemporaryFile("w+") as f:
    f.write("Contenido temporal")
    f.seek(0)
    print(f.read())
```

Explicación:

- `TemporaryFile` crea un fichero que se elimina al cerrarse.
- `seek(0)` mueve el puntero al inicio para poder leer lo escrito.

Comandos explicados: `import tempfile`, `TemporaryFile()`, `seek()`, modos `"w+"`.

Ejercicio 24 – Buscar y reemplazar texto en un fichero

Enunciado: Reemplazar "Hola" por "Adiós" en `archivo1.txt`.

```
with open("archivo1.txt", "r") as f:
    contenido = f.read()

contenido = contenido.replace("Hola", "Adiós")

with open("archivo1.txt", "w") as f:
    f.write(contenido)
```

Explicación:

- `replace()` sustituye las palabras en el string.
- Modo `"w"` sobrescribe el fichero con el contenido modificado.

Comandos explicados: `replace()`, lectura y escritura de ficheros.

Ejercicio 25 – Registrar logs en un fichero

Enunciado: Crear un fichero de logs y escribir mensajes con marca de tiempo.

```
from datetime import datetime


with open("log.txt", "a") as f:
    f.write(f"[{datetime.now()}] Inicio del programa\n")
```

Explicación:

- `datetime.now()` obtiene la fecha y hora actuales.
- Modo `"a"` añade la línea al final del fichero.
- Útil para registrar eventos o errores.

Comandos explicados: `import datetime`, `datetime.now()`, f-strings, modo `a`.

Ejercicio 26 – Leer ficheros de logs línea por línea

Enunciado: Mostrar solo las líneas que contienen la palabra "error" en `log.txt`.

```
with open("log.txt", "r") as f:
    for linea in f:
        if "error" in linea.lower():
            print(linea.strip())
```

Explicación:

- `linea.lower()` permite buscar sin importar mayúsculas/minúsculas.
- `strip()` limpia saltos de línea y espacios.

Comandos explicados: Iteración sobre ficheros, condicionales, `strip()`, `lower()`.

Ejercicio 27 – Dividir un fichero en varios ficheros pequeños

Enunciado: Dividir `archivo1.txt` en ficheros de 2 líneas cada uno.

```
with open("archivo1.txt", "r") as f:
    lineas = f.readlines()

for i in range(0, len(lineas), 2):
    with open(f"parte_{i//2 + 1}.txt", "w") as f_parte:
        f_parte.writelines(lineas[i:i+2])
```

Explicación:

- `readlines()` crea una lista de líneas.
- `writelines()` escribe varias líneas de la lista en otro fichero.

Comandos explicados: `readlines()`, `writelines()`, bucles `for`, slicing de listas.

Ejercicio 28 – Combinar varios ficheros en uno

Enunciado: Combinar `parte_1.txt` y `parte_2.txt` en `combinado.txt`.

```
archivos = ["parte_1.txt", "parte_2.txt"]


with open("combinado.txt", "w") as destino:
    for nombre in archivos:
        with open(nombre, "r") as f:
            destino.write(f.read())
```

Explicación:

- Se itera sobre la lista de ficheros y se escribe su contenido en uno solo.
- Útil para fusionar datos.

Comandos explicados: Iteración, `open()`, `read()`, `write()`.

Ejercicio 29 – Leer CSV con encabezado usando DictReader

Enunciado: Leer `personas.csv` y acceder a cada campo por nombre.

```
import csv

with open("personas.csv", "r") as f:
    reader = csv.DictReader(f)
    for fila in reader:
        print(fila["Nombre"], fila["Edad"])
```

Explicación:

- `DictReader` devuelve cada fila como un diccionario.
- Se accede a cada columna por nombre.

Comandos explicados: `csv.DictReader`, diccionarios, iteración, lectura de CSV.

Ejercicio 30 – Escribir CSV con DictWriter

Enunciado: Crear un CSV usando diccionarios.

```
import csv

datos = [{"Nombre": "Ana", "Edad": 25}, {"Nombre": "Luis", "Edad": 30}]

with open("personas2.csv", "w", newline="") as f:
    campos = ["Nombre", "Edad"]
    writer = csv.DictWriter(f, fieldnames=campos)
    writer.writeheader()
    writer.writerows(datos)
```

Explicación:

- `DictWriter` permite escribir diccionarios en CSV.
- `writeheader()` escribe la cabecera automáticamente.
- `writerows()` escribe varias filas a la vez.

Comandos explicados: `csv.DictWriter`, `writeheader()`, `writerows()`, iteración implícita sobre lista de diccionarios.

Ejercicio 31 – Leer y escribir ficheros usando rutas completas

Enunciado: Crear un fichero en una carpeta específica y leerlo desde su ruta absoluta.

```
import os

ruta = os.path.join("carpeta", "archivo_ruta.txt")
os.makedirs("carpeta", exist_ok=True)

with open(ruta, "w") as f:
    f.write("Contenido con ruta completa")

with open(ruta, "r") as f:
    print(f.read())
```

Explicación:

- `os.path.join()` construye rutas de forma segura.
- `os.makedirs()` crea la carpeta si no existe (`exist_ok=True`).
- Se abre el fichero en modo escritura y luego lectura.

Comandos explicados: `os.path.join()`, `os.makedirs()`, `open()`, modos `w` y `r`, `read()`, `write()`.

Ejercicio 32 – Renombrar un fichero

Enunciado: Renombrar `archivo_ruta.txt` a `archivo_renombrado.txt`.

```
import os


os.rename("carpeta/archivo_ruta.txt", "carpeta/archivo_renombrado.txt")
```

Explicación:

- `os.rename()` cambia el nombre del fichero o carpeta.
- Si el fichero no existe, lanzará un error.

Comandos explicados: `os.rename()`, rutas de ficheros.

Ejercicio 33 – Borrar un fichero

Enunciado: Eliminar el fichero `archivo_renombrado.txt`.

```
import os


os.remove("carpeta/archivo_renombrado.txt")
```

Explicación:

- `os.remove()` elimina el fichero indicado.
- Debe existir, de lo contrario lanza `FileNotFoundError`.

Comandos explicados: `os.remove()`, manejo de ficheros.

Ejercicio 34 – Crear y leer fichero CSV con delimitador personalizado

Enunciado: Crear un CSV separado por `;` y leerlo.

```
import csv

datos = [["Nombre", "Edad"], ["Ana", 25], ["Luis", 30]]

with open("personas3.csv", "w", newline="") as f:
    writer = csv.writer(f, delimiter=";")
    writer.writerows(datos)

with open("personas3.csv", "r") as f:
    reader = csv.reader(f, delimiter=";")
    for fila in reader:
        print(fila)
```

Explicación:

- `delimiter=";"` indica que se usa `;` en lugar de coma.
- Se escribe y se lee el fichero respetando ese delimitador.

Comandos explicados: `csv.writer`, `csv.reader`, `delimiter`, `writerows()`, iteración.

Ejercicio 35 – Comprimir un fichero con gzip

Enunciado: Comprimir `archivo1.txt` en `archivo1.txt.gz`.

```
import gzip


with open("archivo1.txt", "rb") as f_in:
    with gzip.open("archivo1.txt.gz", "wb") as f_out:
        f_out.writelines(f_in)
```

Explicación:

- `gzip.open()` permite escribir un fichero comprimido.
- Se leen las líneas en binario (`rb`) y se escriben comprimidas (`wb`).

Comandos explicados: `gzip.open()`, modos binarios `rb` y `wb`, `writelines()`.

Ejercicio 36 – Leer fichero gzip

Enunciado: Leer el contenido de `archivo1.txt.gz`.

```
import gzip

with gzip.open("archivo1.txt.gz", "rb") as f:
    contenido = f.read()
    print(contenido.decode("utf-8"))
```

Explicación:

- `f.read()` devuelve bytes; se decodifican con `decode("utf-8")`.
- Permite acceder al contenido comprimido.

Comandos explicados: `gzip.open()`, `read()`, `decode()`, modos binarios.

Ejercicio 37 – Copiar fichero usando shutil

Enunciado: Copiar `archivo1.txt` a `copia_shutil.txt`.

```
import shutil


shutil.copy("archivo1.txt", "copia_shutil.txt")
```

Explicación:

- `shutil.copy()` copia el contenido del fichero.
- Mantiene permisos básicos, pero no metadata completa.

Comandos explicados: `import shutil`, `shutil.copy()`, rutas de ficheros.

Ejercicio 38 – Mover fichero con shutil

Enunciado: Mover `copia_shutil.txt` a la carpeta `carpeta`.

```
import shutil
import os

os.makedirs("carpeta", exist_ok=True)
shutil.move("copia_shutil.txt", "carpeta/copia_shutil.txt")
```

Explicación:

- `shutil.move()` mueve o renombra un fichero a otra ubicación.
- Crea la carpeta si no existe.

Comandos explicados: `shutil.move()`, `os.makedirs()`, rutas.

Ejercicio 39 – Contar caracteres, palabras y líneas

Enunciado: Mostrar número de líneas, palabras y caracteres en `archivo1.txt`.

```
with open("archivo1.txt", "r") as f:
    lineas = f.readlines()
    num_lineas = len(lineas)
    num_palabras = sum(len(linea.split()) for linea in lineas)
    num_caracteres = sum(len(linea) for linea in lineas)

print(num_lineas, num_palabras, num_caracteres)
```

Explicación:

- `split()` separa palabras en cada línea.
- `len()` cuenta caracteres o elementos de lista.
- `sum()` suma todos los valores.

Comandos explicados: `readlines()`, `split()`, `len()`, `sum()`, iteración.

Ejercicio 40 – Buscar múltiples palabras en un fichero

Enunciado: Comprobar si "Hola" o "Mundo" aparecen en `archivo1.txt`.

```
palabras = ["Hola", "Mundo"]

with open("archivo1.txt", "r") as f:
    contenido = f.read()
    for palabra in palabras:
        if palabra in contenido:
            print(f"{palabra} encontrada")
```

Explicación:

- Iteramos sobre la lista de palabras y usamos `in` para buscarlas.
- Se imprime mensaje si se encuentra cada palabra.

Comandos explicados: Iteración, operador `in`, `read()`, listas.

Ejercicio 41 – Leer JSON con lista de diccionarios

Enunciado: Leer un fichero `datos.json` que contiene una lista de diccionarios y mostrar cada entrada.

```
import json

with open("datos.json", "r") as f:
    lista = json.load(f)

for item in lista:
    print(f"Nombre: {item['Nombre']}, Edad: {item['Edad']}")
```

Explicación:

- `json.load()` convierte el contenido JSON a objetos Python.
- Iteramos sobre la lista y accedemos a cada diccionario por clave.

Comandos explicados: `json.load()`, iteración, acceso a diccionarios, `with`.

Ejercicio 42 – Guardar lista de diccionarios en JSON

Enunciado: Crear `datos.json` a partir de una lista de diccionarios.

```
import json

lista = [{"Nombre": "Ana", "Edad": 25}, {"Nombre": "Luis", "Edad": 30}]

with open("datos.json", "w") as f:
    json.dump(lista, f, indent=4)
```

Explicación:

- `json.dump()` guarda la lista en formato JSON.
- `indent=4` formatea el JSON para que sea legible.

Comandos explicados: `json.dump()`, `indent`, modo `w`, listas y diccionarios.

Ejercicio 43 – Manejo de excepciones al leer JSON

Enunciado: Leer JSON y controlar error si el fichero no existe.

```
import json


try:
    with open("datos.json", "r") as f:
        datos = json.load(f)
except FileNotFoundError:
    print("El fichero no existe")
except json.JSONDecodeError:
    print("Error al leer JSON")
```

Explicación:

- `FileNotFoundError` captura fichero inexistente.
- `json.JSONDecodeError` captura errores de formato JSON.

Comandos explicados: `try-except`, `json.load()`, manejo de errores específicos.

Ejercicio 44 – Leer y escribir ficheros grandes por bloques

Enunciado: Copiar un fichero grande leyendo bloques de 1024 bytes.

```
with open("archivo_grande.txt", "rb") as origen, open("copia_grande.txt", "wb")
as destino:
    while True:
        bloque = origen.read(1024)
        if not bloque:
            break
        destino.write(bloque)
```

Explicación:

- Se leen bloques pequeños para no cargar todo el fichero en memoria.
- `read(1024)` lee 1024 bytes cada vez.

Comandos explicados: Lectura binaria `rb`, escritura `wb`, bucle `while`, control de flujo.

Ejercicio 45 – Añadir registro con timestamp en CSV

Enunciado: Añadir filas a un CSV con fecha y hora actual.

```
import csv
from datetime import datetime

with open("registro.csv", "a", newline="") as f:
    writer = csv.writer(f)
    writer.writerow([datetime.now(), "Evento realizado"])
```

Explicación:

- `datetime.now()` obtiene la fecha y hora actual.
- Se añade como una fila al CSV existente.

Comandos explicados: `csv.writer()`, `writerow()`, `datetime.now()`, modo `a`.

Ejercicio 46 – Leer ficheros con múltiples tipos de salto de línea

Enunciado: Leer un fichero independientemente del salto de línea usado (`\n`, `\r\n`).

```
with open("archivo_saltos.txt", "r", newline=None) as f:
    for linea in f:
        print(linea.strip())
```

Explicación:

- `newline=None` permite detectar automáticamente distintos tipos de salto de línea.
- `strip()` elimina saltos y espacios.

Comandos explicados: `newline` en `open()`, `strip()`, iteración sobre ficheros.

Ejercicio 47 – Contar frecuencia de palabras

Enunciado: Contar cuántas veces aparece cada palabra en un fichero.

```
from collections import Counter

with open("archivo1.txt", "r") as f:
    palabras = f.read().split()

frecuencia = Counter(palabras)
print(frecuencia)
```

Explicación:

- `split()` separa palabras.
- `Counter` cuenta la frecuencia de cada palabra automáticamente.

Comandos explicados: `collections.Counter`, `split()`, lectura de ficheros.

Ejercicio 48 – Ficheros binarios y struct

Enunciado: Escribir números enteros en binario y leerlos.

```
import struct

numeros = [10, 20, 30]

with open("numeros.bin", "wb") as f:
    for n in numeros:
        f.write(struct.pack("i", n))

with open("numeros.bin", "rb") as f:
    while True:
        bytes_leidos = f.read(4)
        if not bytes_leidos:
            break
        print(struct.unpack("i", bytes_leidos)[0])
```

Explicación:

- `struct.pack()` convierte un entero a bytes.
- `struct.unpack()` convierte los bytes de vuelta a entero.
- Se leen bloques de 4 bytes (tamaño de un entero).

Comandos explicados: `struct.pack()`, `struct.unpack()`, lectura y escritura binaria.

Ejercicio 49 – Filtrar líneas por longitud

Enunciado: Crear un nuevo fichero solo con las líneas que tengan más de 10 caracteres.

```
with open("archivo1.txt", "r") as f_in, open("archivo_filtrado.txt", "w") as f_out:
    for linea in f_in:
        if len(linea.strip()) > 10:
            f_out.write(linea)
```

Explicación:

- `len(linea.strip())` mide longitud sin espacios ni saltos.
- Solo se escriben líneas que cumplan la condición.

Comandos explicados: `len()`, `strip()`, `write()`, iteración sobre ficheros.

Ejercicio 50 – Registrar errores en fichero log

Enunciado: Capturar errores de ejecución y guardarlos en `errores.log`.

```
try:
    x = 10 / 0
except Exception as e:
    with open("errores.log", "a") as f:
        f.write(f"{e}\n")
```

Explicación:

- `Exception as e` captura cualquier error.
- Se añade al log con `write()` y modo `a` para no sobrescribir.

Comandos explicados: Manejo de excepciones, `with open()`, `write()`, modo `a`.

MANUAL DETALLADO DE EJERCICIOS DE MANIPULACIÓN DE CADENAS

Bloque 1: Operaciones Básicas y Acceso (Ejercicios 1-6)

Ejercicio	Enunciado y Explicación	Código de Implementación	Comandos y Controles
1	**Mostrar un mensaje**. Se guarda un texto en la variable `mensaje` y luego se imprime en la pantalla.	`mensaje = "Hola, mundo"print(mensaje)`	**`mensaje`**: variable tipo `str`. **`"Hola, mundo"`**: cadena de texto. **`print()`**: función que muestra un valor en consola.
2	**Concatenar dos cadenas**. Se unen dos cadenas (`nombre` y `apellido`) utilizando el operador `+`, insertando un espacio `" "` entre ellas para formar el `nombre_completo`.	`nombre = "Ana"apellido = "López"nombre_completo = nombre + " " + apellidoprint(nombre_completo)`	**Variables `nombre`, `apellido`**. **Operador `+`**: utilizado para unir cadenas. **`print()`**.
3	**Longitud de una cadena**. El objetivo es mostrar cuántos caracteres tiene la cadena "Programación". La función `len()` cuenta el número de caracteres.	`palabra = "Programación"print("La longitud es:", len(palabra))`	**`len()`**: función que devuelve la longitud de una cadena.
4	**Acceder a un carácter específico**. Muestra la primera letra de la palabra "Python". Las cadenas funcionan como listas de caracteres, y `palabra` accede a la primera posición (índice 0).	`palabra = "Python"print("La primera letra es:", palabra)`	**`[]`**: operador de índice.
5	**Última letra de una cadena**. Muestra la última letra de la palabra "Python". En Python, el uso de índices negativos permite empezar el conteo desde el final, donde **`-1`** representa la última letra.	`palabra = "Python"print("La última letra es:", palabra[-1])`	**Índices negativos** en cadenas.
6	**Subcadena (Slicing)**. Muestra los caracteres del índice 0 al 3 de "Python". La sintaxis `[0:4]` devuelve los caracteres desde la posición inicial (0) hasta justo antes de	`palabra = "Python"print("Subcadena:", palabra[0:4])`	**Slicing (rebanado)**: `[inicio:fin]`.

la posición final (4).

Bloque 2: Métodos de Formato y Modificación (Ejercicios 7-14)

Ejercicio	Enunciado y Explicación	Código de Implementación	Comandos y Controles
7	**Convertir a mayúsculas**. Convierte "python" a mayúsculas. El método **`.upper()`** transforma todos los caracteres a mayúsculas.	`texto = "python"print(texto.upper())`	**Método `.upper()`**.
8	**Convertir a minúsculas**. Convierte "PYTHON" a minúsculas. El método **`.lower()`** transforma todos los caracteres a minúsculas.	`texto = "PYTHON"print(texto.lower())`	**Método `.lower()`**.
9	**Capitalizar**. Convierte "hola mundo" en "Hola mundo". El método **`.capitalize()`** pone en mayúscula solo la primera letra de la cadena, dejando el resto en minúscula.	`texto = "hola mundo"print(texto.capitalize())`	**Método `.capitalize()`**.
10	**Contar apariciones de una letra**. Cuenta cuántas veces aparece "o" en "programacion". El método **`.count("o")`** devuelve la frecuencia de la subcadena especificada.	`texto = "programacion"print("Número de 'o':", texto.count("o"))`	**Método `.count()`**.
11	**Buscar subcadena con `find`**. Busca la posición de "mundo" en "Hola mundo". El método **`.find(subcadena)`** devuelve el índice inicial de la subcadena o -1 si no la encuentra.	`texto = "Hola mundo"posicion = texto.find("mundo")print("Posición de 'mundo':", posicion)`	**`.find(subcadena)`**.
12	**Verificar inicio de cadena**. Comprueba si una cadena empieza con un prefijo específico. **`.startswith(prefijo)`** devuelve `True` si la cadena inicia con el prefijo.	`texto = "Python es genial"print(texto.startswith("Python"))`	**`.startswith(prefijo)`**.
13	**Verificar final de cadena**. Comprueba si una cadena termina con un sufijo específico. **`.endswith(sufijo)`** devuelve `True` si la cadena finaliza con el sufijo.	`texto = "Aprender Python"print(texto.endswith("Python"))`	**`.endswith(sufijo)`**.
14	**Reemplazar palabra**. Cambia "perro" por "gato". El método **`.replace(viejo,`**	`texto = "El perro corre"nuevo_texto = texto.replace("perro",`	**`.replace(viejo, nuevo)`**.

	`nuevo)` sustituye todas las apariciones de la subcadena vieja por la nueva.	`"gato")print(nuevo_texto)`	

Bloque 3: División, Unión y Limpieza (Ejercicios 15-20)

Ejercicio	Enunciado y Explicación	Código de Implementación	Comandos y Controles
15	**Separar palabras con `split`**. Separa una cadena en una lista de subcadenas usando un separador (`","`). **`.split(separador)`** devuelve una lista.	`texto = "uno,dos,tres"palabras = texto.split(",")print(palabras)`	**`.split(separador)`**.
16	**Unir lista en cadena con `join`**. Une elementos de una lista (`["rojo", "verde", "azul"]`) en una sola cadena usando un conector (`"-"`). La sintaxis es `separador.join(lista)`.	`colores = ["rojo", "verde", "azul"]texto = "-".join(colores)print(texto)`	**`.join(iterable)`**: une elementos de una lista.
17	**Eliminar espacios al inicio y fin**. Quita los espacios en blanco sobrantes. **`.strip()`** limpia los espacios tanto a la izquierda como a la derecha de la cadena.	`texto = " hola "print(texto.strip())`	**`.strip()`**: limpia espacios en blanco.
18	**Convertir primera letra de cada palabra en mayúscula**. **`.title()`** convierte la primera letra de *cada palabra* dentro de la cadena a mayúscula.	`texto = "hola mundo en python"print(texto.title())`	**`.title()`**.
19	**Repetir una cadena**. Repite "Python " tres veces. El **operador `*`** permite la repetición de cadenas un número determinado de veces.	`texto = "Python "print(texto * 3)`	**Operador `*`** con cadenas.
20	**Verificar si contiene solo números**. Comprueba si "12345" solo contiene dígitos. **`.isdigit()`** devuelve `True` si todos los caracteres son números.	`texto = "12345"print(texto.isdigit())`	**`.isdigit()`**.

Bloque 4: Verificación de Contenido (Ejercicios 21-26)

Ejercicio	Enunciado y Explicación	Código de Implementación	Comandos y Controles
21	**Verificar si contiene solo letras**. Comprueba si "Python" contiene únicamente caracteres alfabéticos (sin espacios ni números). **`.isalpha()`**.	`texto = "Python"print(texto.isalpha())`	**`.isalpha()`**.
22	**Verificar si contiene letras y números**. Verifica si	`texto = "Python3"print(texto.isal`	**`.isalnum()`**.

	"Python3" contiene una combinación de letras y números. **`.isalnum()`** devuelve `True` si la cadena contiene solo letras y números.	`num())`	
23	**Verificar si es mayúscula**. Comprueba si la cadena está completamente en mayúsculas. **`.isupper()`**.	`texto = "PYTHON"print(texto.isupp er())`	**`.isupper()`**.
24	**Verificar si es minúscula**. Comprueba si la cadena está completamente en minúsculas. **`.islower()`**.	`texto = "python"print(texto.islow er())`	**`.islower()`**.
25	**Contar vocales**. Se utiliza una **comprensión de generador** y la función `sum()` para recorrer la cadena y contar (sumar 1) cada letra que se encuentre dentro de la lista de `vocales` definida.	`texto = "Programación"vocales = "aeiouAEIOU"contador = sum(1 for letra in texto if letra in vocales)print("Número de vocales:", contador)`	**`for` con comprensión**. **`in`**: para verificar pertenencia. **`sum()`**: para acumular la cuenta.
26	**Invertir una cadena**. Invierte la palabra "Python". Se utiliza **Slicing con paso negativo** `[::-1]`, que recorre la cadena de principio a fin, pero en sentido inverso.	`texto = "Python"print("Invertido: ", texto[::-1])`	**Slicing con paso negativo**.

Bloque 5: Alineación y Palíndromos Simples (Ejercicios 27-32)

Ejercicio	Enunciado y Explicación	Código de Implementación	Comandos y Controles
27	**Eliminar solo espacios de la izquierda**. Quita los espacios del inicio. El método **`.lstrip()`** elimina solo los espacios a la izquierda (left strip).	`texto = " hola"print(texto.lstrip ())`	**`.lstrip()`**.
28	**Eliminar solo espacios de la derecha**. Quita los espacios del final. El método **`.rstrip()`** elimina solo los espacios a la derecha (right strip).	`texto = "hola "print(texto.rstrip())`	**`.rstrip()`**.
29	**Centrar texto**. Centra "Python" en un campo de 20 caracteres. **`.center(ancho, relleno)`** coloca la palabra en el centro y rellena los espacios restantes con el carácter especificado (guiones, en este caso).	`texto = "Python"print(texto.cen ter(20, "-"))`	**`.center(ancho, relleno)`**.

30	**Alinear texto a la derecha**. Muestra "Python" alineado a la derecha en 20 espacios. **`.rjust(ancho)`** rellena con espacios a la izquierda para alinear el texto a la derecha.	`texto = "Python"print(texto.rjust(20))`	**`.rjust(ancho)`**.
31	**Verificar si una palabra es palíndromo**. Compara la cadena original con su versión invertida (`texto[::-1]`).	`texto = "radar"if texto == texto[::-1]:print("Es un palíndromo")else:print("No es un palíndromo")`	**`if/else`**. **Slicing con paso negativo** (`[::-1]`).
32	**Contar palabras en una frase**. Se usa **`.split()`** sin argumento para dividir la frase en una lista de palabras (usando espacios como separadores) y luego se usa `len()` para contar cuántos elementos hay en la lista.	`frase = "Hola mundo en Python"palabras = frase.split()print("Número de palabras:", len(palabras))`	**`.split()`**. **`len()`**.

Bloque 6: Códigos ASCII y Frecuencia (Ejercicios 33-37)

Ejercicio	Enunciado y Explicación	Código de Implementación	Comandos y Controles
33	**Obtener códigos ASCII de cada letra**. **`ord(carácter)`** devuelve el valor ASCII (numérico) de un carácter. Se utiliza una **comprensión de listas** para aplicar `ord()` a cada letra.	`texto = "ABC"ascii_codes = [ord(letra) for letra in texto]print(ascii_codes)`	**`ord(carácter)`**. **Comprensión de listas**.
34	**Convertir códigos ASCII en texto**. **`chr(número)`** transforma un número ASCII en su carácter correspondiente. Se utiliza **`.join()`** para unir los caracteres resultantes.	`codigos =texto = "".join(chr(c) for c in codigos)print(texto)`	**`chr(número)`**. **`.join()`**.
35	**Contar cuántas veces aparece cada letra**. Se utiliza **`set(texto)`** para obtener solo las letras únicas. Luego, se usa el **Diccionario por comprensión** y el método **`.count()`** para registrar la frecuencia de cada letra única.	`texto = "banana"frecuencia = {letra: texto.count(letra) for letra in set(texto)} print(frecuencia)`	**`set()`**. **`.count()`**. **Diccionario por comprensión**.
36	**Eliminar caracteres no alfabéticos**. Se recorre la cadena filtrando los caracteres. **`.isalpha()`** detecta si un carácter es una letra, y **`.join()`** une solo esos caracteres filtrados.	`texto = "H0l@ Pyth0n!"solo_letras = "".join(c for c in texto if c.isalpha())print(solo_letras)`	**`.isalpha()`**. **`.join()`**.
37	**Convertir cada palabra en**	`frase = "python es`	**`.split()`**, **`.up`**

	mayúscula. La frase se separa en palabras usando **`.split()`**, cada palabra se convierte a mayúscula con **`.upper()`** y finalmente se vuelven a unir con espacios usando **`.join()`**.	`genial"mayusculas = " ".join(palabra.upper() for palabra in frase.split())print(ma yusculas)`	**`per()`**, **`.join( )`**.

Bloque 7: Filtros y Transformaciones Avanzadas (Ejercicios 38-43)

Ejercicio	Enunciado y Explicación	Código de Implementación	Comandos y Controles
38	**Eliminar vocales de un texto**. Se recorre el texto y se filtran los caracteres que **`not in`** (no están en) la cadena de vocales definida. **`.join()`** une los caracteres restantes.	`texto = "Programación"vocales = "aeiouAEIOU"sin_vocale s = "".join(c for c in texto if c not in vocales)print(sin_voca les)`	**`not in`**. **`.join()`**.
39	**Invertir cada palabra**. Se usa **`.split()`** para obtener palabras, cada palabra se invierte con el **Slicing `[::-1]`** y se unen con **`.join()`** usando un espacio como separador.	`frase = "Hola mundo"invertida = " ".join(p[::-1] for p in frase.split())print(in vertida)`	**`.split()`**. **Slicing `[::-1]`**. **`.join()`**.
40	**Formatear cadenas con f-strings**. Los **f-strings** (cadenas literales formateadas) permiten insertar variables (`nombre`, `edad`) directamente dentro de la cadena usando llaves `{}`.	`nombre = "Ana"edad = 25print(f"Mi nombre es {nombre} y tengo {edad} años")`	**`f"texto {variable}"`**.
41	**Eliminar caracteres repetidos**. **`dict.fromkeys( texto)`** crea un diccionario donde cada carácter único de la cadena se convierte en una clave (eliminando duplicados). **`.join ()`** convierte estas	`texto = "programacion"sin_repe tidos = "".join(dict.fromkeys( texto))`	**`dict.fromkeys()`**. **`.join( )`**.

	claves únicas de nuevo en una cadena.		
42	**Cifrado César (desplazar letras)**. Se utiliza **`ord()`** para obtener el valor ASCII, se realiza la operación matemática de desplazamiento (`% 26` asegura que se mantenga dentro del alfabeto), y **`chr()`** devuelve el carácter cifrado.	`texto = "abc"desplazamiento = 3cifrado = "".join(chr((ord(c) - 97 + desplazamiento) % 26 + 97) for c in texto)`	**`ord()`**, **`chr()`**. **Operadores** `%` (módulo), `+` (suma).
43	**Quitar acentos de un texto**. Requiere el módulo `unicodedata`. Se normaliza el texto (`"NFD"`) para separar los caracteres de sus acentos. Luego se filtran aquellos caracteres que tienen una categoría de marca diacrítica ("Mn") mediante **`.category()`**.	`import unicodedatatexto = "canción"normalizado = "".join(c for c in unicodedata.normalize( "NFD", texto) if unicodedata.category(c ) != "Mn")`	**`unicodedata.normalize( ).category()`**.

Bloque 8: Palíndromos, Frecuencia de Palabras y Acrónimos (Ejercicios 44-48)

Ejercicio	Enunciado y Explicación	Código de Implementación	Comandos y Controles
44	**Detectar palíndromos en frases**. Para ignorar espacios y mayúsculas, se usa **`.lower()`** para convertir a minúsculas, **`.split()`** para dividir la frase, y **`.join()`** para unirla sin espacios. Finalmente, se compara la cadena resultante con su inversa (`[::-1]`).	`frase = "Anita lava la tina"procesada = "".join(frase.lower().split())if procesada == procesada[::-1]:print(" Es palíndromo")else:print( "No es palíndromo")`	**`.lower()`**, **`.split()`**, **`.join( )`**. **Slicing `[::-1]`**.
45	**Contar frecuencia de palabras**. Se usa **`.split()`** para obtener la lista de palabras. Se aplica **`set()`** para obtener las palabras únicas. Luego, el **Diccionario por comprensión** utiliza **`.count()`** para determinar la	`frase = "hola mundo hola"palabras = frase.split()frecuencia = {p: palabras.count(p) for p in set(palabras)} print(frecuencia)`	**`.split()`**, **`set()`**. **Diccionario por comprensión**.

	frecuencia de cada palabra única en la lista original.		
46	**Generar acrónimo**. Se divide la frase con **`.split()`**. Se recorre la lista resultante, y si la longitud de la palabra es mayor a 2 (usando una **condición `if` en la comprensión**), se toma la primera letra (`p`) y se convierte a mayúscula (`.upper()`), uniéndose con **`.join()`**.	`frase = "Asociación Nacional de Programadores"acronimo = "".join(p.upper() for p in frase.split() if len(p) > 2)`	**`.split()`**, **`.upper()`**. **Condición `if` en comprensión**.
47	**Insertar separador cada 3 caracteres**. Se utiliza la función **`range(inicio, fin, paso)`** para generar índices que saltan de 3 en 3. Con **Slicing con paso fijo** (`texto[i:i+3]`) se agrupan los caracteres de tres en tres, y luego se unen con **`.join()`** usando el guion como separador.	`texto = "123456789"separado = "-".join(texto[i:i+3] for i in range(0, len(texto), 3))`	**`range(inicio, fin, paso)`**. **Slicing con paso fijo**. **`.join()`**.
48	**Encontrar palabra más larga**. Se usa **`.split()`** para obtener palabras. La función **`max(iterable, key=función)`** encuentra el elemento más grande de la lista, utilizando la función `len` para determinar la longitud de cada palabra (la "clave" de comparación).	`frase = "Python es increíble"palabra_larga = max(frase.split(), key=len)print(palabra_larga)`	**`.split()`**. **`max(iterable, key=función)`**.

Bloque 9: Funcionalidad Externa y Justificación (Ejercicios 49-50)

Ejercicio	Enunciado y Explicación	Código de Implementación	Comandos y Controles
49	**Generar contraseñas aleatorias**. Se requiere importar los módulos **`random`** y **`string`**. Se define la fuente de caracteres usando `string.ascii_letters` y `string.digits`. Se usa **`random.choice()`** para elegir 8 caracteres al azar, que luego se unen con **`.join()`**.	`import randomimport stringcaracteres = string.ascii_letters + string.digitspassword = "".join(random.choice(caracteres) for _ in range(8))`	**`random.choice()`**. **`string.ascii_letters`**, **`string.digits`**.
50	**Justificar texto en párrafos**. El método **`.ljust(ancho, relleno)`** alinea la frase a la izquierda en un campo de	`frase = "Python es divertido"justificado = frase.ljust(30, ".")print(justificado)`	**`.ljust(ancho, relleno)`**.

30 caracteres, rellenando el espacio sobrante con el carácter especificado (puntos).

Ejercicio 1: Mostrar un mensaje

Enunciado:
Escribe un programa que muestre en pantalla la cadena `"Hola, mundo"`.

Código:

```
mensaje = "Hola, mundo"
print(mensaje)
```

Explicación:
Se guarda un texto en la variable `mensaje` y luego se imprime en la pantalla.

Comandos y variables:

- `mensaje`: variable tipo `str`.
- `"Hola, mundo"`: cadena de texto.
- `print()`: función que muestra un valor en consola.

Ejercicio 2: Concatenar dos cadenas

Enunciado:
Crea un programa que una tu nombre y apellido en una sola cadena.

Código:

```
nombre = "Ana"
apellido = "López"
nombre_completo = nombre + " " + apellido
print(nombre_completo)
```

Explicación:
Se concatenan dos cadenas con el operador `+`, agregando un espacio `" "` entre ellas.

Comandos:

- Variables `nombre`, `apellido`.
- Operador `+` para unir cadenas.
- `print()` para mostrar resultado.

Ejercicio 3: Longitud de una cadena

Enunciado:
Muestra cuántos caracteres tiene la cadena `"Programación"`.

Código:

```
palabra = "Programación"
print("La longitud es:", len(palabra))
```

Explicación:
`len()` cuenta el número de caracteres de la cadena.

Comandos:

- `len()`: función que devuelve longitud de una cadena.

Ejercicio 4: Acceder a un carácter específico

Enunciado:
Muestra la primera letra de la palabra `"Python"`.

Código:

```
palabra = "Python"
print("La primera letra es:", palabra[0])
```

Explicación:
Las cadenas funcionan como listas de caracteres. `palabra[0]` devuelve la primera letra.

Comandos:

- `[]`: operador de índice.

Ejercicio 5: Última letra de una cadena

Enunciado:
Muestra la última letra de la palabra `"Python"`.

Código:

```
palabra = "Python"
print("La última letra es:", palabra[-1])
```

Explicación:
En Python, los índices negativos empiezan desde el final: `-1` es la última letra.

Comandos:

- Índices negativos en cadenas.

Ejercicio 6: Subcadena

Enunciado:
Muestra los caracteres del índice 0 al 3 de la palabra `"Python"`.

Código:

```
palabra = "Python"
print("Subcadena:", palabra[0:4])
```

Explicación:
`palabra[0:4]` devuelve los caracteres desde la posición 0 hasta la 3 (el 4 no se incluye).

Comandos:

- Slicing (rebanado): `[inicio:fin]`.

Ejercicio 7: Convertir a mayúsculas

Enunciado:
Convierte `"python"` a mayúsculas.

Código:

```
texto = "python"
print(texto.upper())
```

Explicación:
`.upper()` convierte todos los caracteres a mayúsculas.

Comandos:

- Método `.upper()`.

Ejercicio 8: Convertir a minúsculas

Enunciado:
Convierte `"PYTHON"` a minúsculas.

Código:

```
texto = "PYTHON"
print(texto.lower())
```

Explicación:
`.lower()` convierte todos los caracteres a minúsculas.

Comandos:

- Método `.lower()`.

Ejercicio 9: Capitalizar

Enunciado:
Convierte `"hola mundo"` en `"Hola mundo"`.

Código:

```
texto = "hola mundo"
print(texto.capitalize())
```

Explicación:
`.capitalize()` pone en mayúscula la primera letra y el resto en minúscula.

Comandos:

- Método `.capitalize()`.

Ejercicio 10: Contar apariciones de una letra

Enunciado:
Cuenta cuántas veces aparece `"o"` en `"programacion"`.

Código:

```
texto = "programacion"
print("Número de 'o':", texto.count("o"))
```

Explicación:
`.count("o")` devuelve cuántas veces aparece la letra `"o"`.

Comandos:

- Método `.count()`.

Ejercicio 11: Buscar subcadena con `find`

Enunciado:
Busca la posición de la palabra `"mundo"` en `"Hola mundo"`.

Código:

```
texto = "Hola mundo"

posicion = texto.find("mundo")

print("Posición de 'mundo':", posicion)
```

Explicación:
`.find("mundo")` devuelve el índice donde empieza `"mundo"`.

Comandos:

- `.find(subcadena)`: busca y devuelve índice o `-1` si no lo encuentra.

Ejercicio 12: Verificar inicio de cadena

Enunciado:
Comprueba si `"Python es genial"` empieza con `"Python"`.

Código:

```
texto = "Python es genial"
print(texto.startswith("Python"))
```

Explicación:
`.startswith()` devuelve `True` si la cadena empieza con la subcadena.

Comandos:

- `.startswith(prefijo)`.

Ejercicio 13: Verificar final de cadena

Enunciado:
Comprueba si `"Aprender Python"` termina con `"Python"`.

Código:

```
texto = "Aprender Python"
print(texto.endswith("Python"))
```

Explicación:
`.endswith()` devuelve `True` si la cadena termina con la subcadena.

Comandos:

- `.endswith(sufijo)`.

Ejercicio 14: Reemplazar palabra

Enunciado:
Cambia `"perro"` por `"gato"` en `"El perro corre"`.

Código:

```
texto = "El perro corre"

nuevo_texto = texto.replace("perro", "gato")

print(nuevo_texto)
```

Explicación:
`.replace()` sustituye todas las apariciones de una subcadena.

Comandos:

- `.replace(viejo, nuevo)`.

Ejercicio 15: Separar palabras con `split`

Enunciado:
Separa `"uno,dos,tres"` en una lista de palabras.

Código:

```
texto = "uno,dos,tres"

palabras = texto.split(",")

print(palabras)
```

Explicación:
`.split(",")` divide la cadena usando `,` como separador.

Comandos:

- `.split(separador)`: devuelve lista de subcadenas.

Ejercicio 16: Unir lista en cadena con `join`

Enunciado:
Une `["rojo", "verde", "azul"]` en `"rojo-verde-azul"`.

Código:

```
colores = ["rojo", "verde", "azul"]

texto = "-".join(colores)

print(texto)
```

Explicación:
`"-".join(lista)` une los elementos separados por `-`.

Comandos:

- `.join(iterable)`: une elementos de una lista en una sola cadena.

Ejercicio 17: Eliminar espacios al inicio y fin

Enunciado:
Quita los espacios de `" hola "`.

Código:

```
texto = "   hola   "
print(texto.strip())
```

Explicación:
`.strip()` elimina espacios al inicio y final de la cadena.

Comandos:

- `.strip()`: limpia espacios en blanco.

Ejercicio 18: Convertir primera letra de cada palabra en mayúscula

Enunciado:
Convierte `"hola mundo en python"` en `"Hola Mundo En Python"`.

Código:

```
texto = "hola mundo en python"

print(texto.title())
```

Explicación:
`.title()` convierte la primera letra de cada palabra en mayúscula.

Comandos:

- `.title()`.

Ejercicio 19: Repetir una cadena

Enunciado:
Repite `"Python "` tres veces.

Código:

```
texto = "Python "
print(texto * 3)
```

Explicación:
El operador `*` permite repetir cadenas un número determinado de veces.

Comandos:

- Operador `*` con cadenas.

Ejercicio 20: Verificar si contiene solo números

Enunciado:
Comprueba si `"12345"` contiene solo números.

Código:

```
texto = "12345"

print(texto.isdigit())
```

Explicación:
`.isdigit()` devuelve `True` si todos los caracteres son números.

Comandos:

- `.isdigit()`.

Ejercicio 21: Verificar si contiene solo letras

Enunciado:
Comprueba si `"Python"` contiene únicamente letras.

Código:

```
texto = "Python"

print(texto.isalpha())
```

Explicación:
`.isalpha()` devuelve `True` si todos los caracteres son letras (sin espacios ni números).

Comandos:

- `.isalpha()`

Ejercicio 22: Verificar si contiene letras y números

Enunciado:
Verifica si `"Python3"` contiene letras y números.

Código:

```
texto = "Python3"

print(texto.isalnum())
```

Explicación:
`.isalnum()` devuelve `True` si la cadena tiene solo letras y números.

Comandos:

- `.isalnum()`

Ejercicio 23: Verificar si es mayúscula

Enunciado:
Comprueba si `"PYTHON"` está completamente en mayúsculas.

Código:

```
texto = "PYTHON"

print(texto.isupper())
```

Explicación:
`.isupper()` devuelve `True` si todos los caracteres son mayúsculas.

Comandos:

- `.isupper()`

Ejercicio 24: Verificar si es minúscula

Enunciado:
Comprueba si `"python"` está completamente en minúsculas.

Código:

```
texto = "python"

print(texto.islower())
```

Explicación:
`.islower()` devuelve `True` si todos los caracteres son minúsculas.

Comandos:

- `.islower()`

Ejercicio 25: Contar vocales

Enunciado:
Cuenta cuántas vocales tiene la palabra `"Programación"`.

Código:

```
texto = "Programación"
vocales = "aeiouAEIOU"
contador = sum(1 for letra in texto if letra in vocales)
print("Número de vocales:", contador)
```

Explicación:
Se recorre la cadena y se cuenta cada letra que esté en `"aeiouAEIOU"`.

Comandos:

- `for` con comprensión.
- `in` para verificar pertenencia.
- `sum()` para acumular.

Ejercicio 26: Invertir una cadena

Enunciado:
Invierte la palabra `"Python"` para que quede `"nohtyP"`.

Código:

```
texto = "Python"
print("Invertido:", texto[::-1])
```

Explicación:
`[::-1]` recorre la cadena desde el final hacia el inicio.

Comandos:

- Slicing con paso negativo.

Ejercicio 27: Eliminar solo espacios de la izquierda

Enunciado:
Quita los espacios del inicio en `"  hola"`.

Código:

```
texto = "   hola"
print(texto.lstrip())
```

Explicación:
`.lstrip()` elimina solo los espacios a la izquierda.

Comandos:

- `.lstrip()`

Ejercicio 28: Eliminar solo espacios de la derecha

Enunciado:
Quita los espacios del final en `"hola "`.

Código:

```
texto = "hola   "
print(texto.rstrip())
```

Explicación:
`.rstrip()` elimina solo los espacios a la derecha.

Comandos:

- `.rstrip()`

Ejercicio 29: Centrar texto

Enunciado:
Centrar `"Python"` en un campo de 20 caracteres usando guiones.

Código:

```
texto = "Python"
print(texto.center(20, "-"))
```

Explicación:
`.center(20, "-")` coloca la palabra en el centro y rellena con guiones a ambos lados.

Comandos:

- `.center(ancho, relleno)`

Ejercicio 30: Alinear texto a la derecha

Enunciado:
Muestra `"Python"` alineado a la derecha en 20 espacios.

Código:

```
texto = "Python"
print(texto.rjust(20))
```

Explicación:
`.rjust(20)` coloca la palabra a la derecha, rellenando con espacios a la izquierda.

Comandos:

- `.rjust(ancho)`

Ejercicio 31: Verificar si una palabra es palíndromo

Enunciado:
Escribe un programa que verifique si `"radar"` es un palíndromo (se lee igual al derecho y al revés).

Código:

```
texto = "radar"
if texto == texto[::-1]:
    print("Es un palíndromo")
else:
    print("No es un palíndromo")
```

Explicación:
Se compara la cadena original con su versión invertida (`[::-1]`).

Comandos:

- `if/else`
- Slicing con paso negativo.

Ejercicio 32: Contar palabras en una frase

Enunciado:
Cuenta cuántas palabras tiene `"Hola mundo en Python"`.

Código:

```
frase = "Hola mundo en Python"

palabras = frase.split()

print("Número de palabras:", len(palabras))
```

Explicación:
`.split()` divide la frase en palabras y `len()` cuenta cuántas son.

Comandos:

- `.split()`
- `len()`

Ejercicio 33: Obtener códigos ASCII de cada letra

Enunciado:
Convierte `"ABC"` en sus códigos ASCII.

Código:

```
texto = "ABC"

ascii_codes = [ord(letra) for letra in texto]

print(ascii_codes)
```

Explicación:
`ord()` devuelve el valor ASCII de un carácter.

Comandos:

- `ord(carácter)`
- Comprensión de listas.

Ejercicio 34: Convertir códigos ASCII en texto

Enunciado:
Convierte `[65, 66, 67]` en `"ABC"`.

Código:

```
codigos = [65, 66, 67]

texto = "".join(chr(c) for c in codigos)

print(texto)
```

Explicación:
`chr()` transforma un número ASCII en carácter.

Comandos:

- `chr(número)`
- `.join()`

Ejercicio 35: Contar cuántas veces aparece cada letra

Enunciado:
Cuenta la frecuencia de cada letra en `"banana"`.

Código:

```
texto = "banana"
frecuencia = {letra: texto.count(letra) for letra in set(texto)}
print(frecuencia)
```

Explicación:
Se usa un `set` para obtener letras únicas y `.count()` para contarlas.

Comandos:

- `set()`
- `.count()`
- Diccionario por comprensión.

Ejercicio 36: Eliminar caracteres no alfabéticos

Enunciado:
De `"H0l@ Pyth0n!"` conserva solo letras.

Código:

```
texto = "H0l@ Pyth0n!"
solo_letras = "".join(c for c in texto if c.isalpha())
print(solo_letras)
```

Explicación:
`.isalpha()` detecta si un carácter es letra.

Comandos:

- `.isalpha()`
- `.join()`

Ejercicio 37: Convertir cada palabra en mayúscula

Enunciado:
Convierte `"python es genial"` en `"PYTHON ES GENIAL"`.

Código:

```
frase = "python es genial"

mayusculas = " ".join(palabra.upper() for palabra in frase.split())

print(mayusculas)
```

Explicación:
Se separa la frase en palabras, cada una se convierte a mayúscula y se vuelve a unir.

Comandos:

- `.split()`
- `.upper()`
- `.join()`

Ejercicio 38: Eliminar vocales de un texto

Enunciado:
Quita todas las vocales de `"Programación"`.

Código:

```
texto = "Programación"
vocales = "aeiouAEIOU"
sin_vocales = "".join(c for c in texto if c not in vocales)
print(sin_vocales)
```

Explicación:
Se recorre el texto y se filtran los caracteres que no son vocales.

Comandos:

- `not in`
- `.join()`

Ejercicio 39: Invertir cada palabra

Enunciado:
Convierte `"Hola mundo"` en `"aloH odnum"`.

Código:

```
frase = "Hola mundo"
invertida = " ".join(p[::-1] for p in frase.split())
print(invertida)
```

Explicación:
Cada palabra se invierte con slicing y luego se unen con espacios.

Comandos:

- `.split()`
- Slicing `[::-1]`
- `.join()`

Ejercicio 40: Formatear cadenas con f-strings

Enunciado:
Muestra `"Mi nombre es Ana y tengo 25 años"`.

Código:

```
nombre = "Ana"
edad = 25
print(f"Mi nombre es {nombre} y tengo {edad} años")
```

Explicación:
Los f-strings permiten insertar variables directamente en la cadena.

Comandos:

- `f"texto {variable}"`

Ejercicio 41: Eliminar caracteres repetidos

Enunciado:
Deja solo una vez cada letra en `"programacion"`.

Código:

```
texto = "programacion"

sin_repetidos = "".join(dict.fromkeys(texto))

print(sin_repetidos)
```

Explicación:
`dict.fromkeys()` elimina duplicados porque las claves de un diccionario son únicas.

Comandos:

- `dict.fromkeys()`
- `.join()`

Ejercicio 42: Cifrado César (desplazar letras)

Enunciado:
Cifra `"abc"` desplazando cada letra 3 posiciones → `"def"`.

Código:

```
texto = "abc"
desplazamiento = 3
cifrado = "".join(chr((ord(c) - 97 + desplazamiento) % 26 + 97) for c in texto)
print(cifrado)
```

Explicación:
Se convierte cada letra en ASCII (`ord`), se aplica un desplazamiento y se regresa con `chr`.

Comandos:

- `ord()`, `chr()`
- Operadores: %, +

Ejercicio 43: Quitar acentos de un texto

Enunciado:
Transforma `"canción"` en `"cancion"`.

Código:

```
import unicodedata


texto = "canción"

normalizado = "".join(

    c for c in unicodedata.normalize("NFD", texto) if unicodedata.category(c) != "Mn"

)

print(normalizado)
```

Explicación:
Se normaliza el texto eliminando las marcas de acentos (`Mn`).

Comandos:

- `unicodedata.normalize()`
- `.category()`

Ejercicio 44: Detectar palíndromos en frases

Enunciado:
Verifica si `"Anita lava la tina"` es un palíndromo (ignora espacios y mayúsculas).

Código:

```
frase = "Anita lava la tina"
procesada = "".join(frase.lower().split())
if procesada == procesada[::-1]:
    print("Es palíndromo")
else:
    print("No es palíndromo")
```

Explicación:
Se eliminan espacios, se pasa a minúsculas y se compara con la inversa.

Comandos:

- `.lower()`, `.split()`, `.join()`
- Slicing `[::-1]`

Ejercicio 45: Contar frecuencia de palabras

Enunciado:
Cuenta cuántas veces aparece cada palabra en `"hola mundo hola"`.

Código:

```
frase = "hola mundo hola"
palabras = frase.split()
frecuencia = {p: palabras.count(p) for p in set(palabras)}
print(frecuencia)
```

Explicación:
`set()` obtiene palabras únicas y `.count()` las cuenta.

Comandos:

- `.split()`
- `set()`
- Diccionario por comprensión

Ejercicio 46: Generar acrónimo

Enunciado:
De `"Asociación Nacional de Programadores"` crea `"ANP"`.

Código:

```
frase = "Asociación Nacional de Programadores"
acronimo = "".join(p[0].upper() for p in frase.split() if len(p) > 2)
print(acronimo)
```

Explicación:
Se toma la primera letra de cada palabra (excluyendo palabras cortas como “de”).

Comandos:

- `.split()`
- `.upper()`
- Condición `if` en comprensión

Ejercicio 47: Insertar separador cada 3 caracteres

Enunciado:
Transforma `"123456789"` en `"123-456-789"`.

Código:

```
texto = "123456789"
separado = "-".join(texto[i:i+3] for i in range(0, len(texto), 3))
print(separado)
```

Explicación:
Se agrupan los caracteres de 3 en 3 con slicing.

Comandos:

- `range(inicio, fin, paso)`
- Slicing con paso fijo
- `.join()`

Ejercicio 48: Encontrar palabra más larga

Enunciado:
De `"Python es increíble"` muestra la palabra más larga.

Código:

```
frase = "Python es increíble"

palabra_larga = max(frase.split(), key=len)

print(palabra_larga)
```

Explicación:
`max()` con `key=len` devuelve la palabra de mayor longitud.

Comandos:

- `.split()`
- `max(iterable, key=función)`

Ejercicio 49: Generar contraseñas aleatorias

Enunciado:
Crea una contraseña de 8 caracteres al azar con letras y números.

Código:

```
import random
import string

caracteres = string.ascii_letters + string.digits
password = "".join(random.choice(caracteres) for _ in range(8))
print(password)
```

Explicación:
Se eligen 8 caracteres aleatorios de letras y números.

Comandos:

- `random.choice()`
- `string.ascii_letters`, `string.digits`

Ejercicio 50: Justificar texto en párrafos

Enunciado:
Justifica la frase `"Python es divertido"` en 30 caracteres con relleno.

Código:

```
frase = "Python es divertido"

justificado = frase.ljust(30, ".")

print(justificado)
```

Explicación:
`.ljust(30, ".")` coloca la frase alineada a la izquierda y rellena con puntos.

Comandos:

- `.ljust(ancho, relleno)`

MANUAL DETALLADO DE EJERCICIOS : MANEJO DE FECHA Y HORA EN PYTHON

Ejercicio 1

Enunciado: Imprime la fecha y hora actuales del sistema. **Solución:**

```
import datetime
ahora = datetime.datetime.now()
print("Fecha y hora actuales:", ahora)
```

Explicación Minuciosa: Para obtener la información de fecha y hora se debe **importar el módulo `datetime`**. La función clave es `datetime.datetime.now()`, que obtiene la fecha y hora actuales del sistema en el momento de la ejecución. Finalmente, se utiliza `print()` para mostrar el resultado en la pantalla. **Comandos y Controles Usados:**

- `import datetime`: Importa el módulo necesario para manejar fechas y horas.
- `datetime.datetime.now()`: Obtiene la fecha y hora actuales.
- `print()`: Imprime el resultado en pantalla.

Ejercicio 2

Enunciado: Muestra solo la fecha actual. **Solución:**

```
import datetime
fecha = datetime.date.today()
print("Fecha actual:", fecha)
```

Explicación Minuciosa: Este ejercicio se enfoca en obtener exclusivamente la fecha (año, mes, día). Se logra utilizando el comando **`datetime.date.today()`**. **Comandos y Controles Usados:**

- `datetime.date.today()`: Obtiene solo la fecha actual (año, mes, día).

Ejercicio 3

Enunciado: Muestra solo la hora actual en formato de 24 horas. **Solución:**

```
import datetime
hora = datetime.datetime.now().time()
print("Hora actual:", hora)
```

Explicación Minuciosa: Se obtiene primero el objeto completo de fecha y hora usando `datetime.datetime.now()`. Luego, se utiliza el método **`.time()` para extraer únicamente la porción correspondiente a la hora** de ese objeto `datetime`. **Comandos y Controles Usados:**

- `datetime.datetime.now()`: Obtiene la fecha y hora actuales.
- `.time()`: Extrae solo la hora.
- `print()`: Imprime en pantalla.

Ejercicio 4

Enunciado: Imprime la fecha actual en formato dd/mm/yyyy. **Solución:**

```
import datetime
ahora = datetime.datetime.now()
fecha_formato = ahora.strftime("%d/%m/%Y")
print("Fecha formateada:", fecha_formato)
```

Explicación Minuciosa: Para aplicar un formato personalizado a la fecha, se usa el método **`strftime()`**. Este método convierte la fecha a una cadena de texto legible según los códigos de formato que se le indiquen. Para el formato solicitado, se usan códigos específicos para día, mes y año. **Comandos y Controles Usados:**

- `strftime()`: Convierte la fecha a un formato legible.
- `%d`: Día con dos dígitos.
- `%m`: Mes con dos dígitos.
- `%Y`: Año con cuatro dígitos.

Ejercicio 5

Enunciado: Imprime la hora actual en formato HH:MM:SS. **Solución:**

```
import datetime
ahora = datetime.datetime.now()
hora_formato = ahora.strftime("%H:%M:%S")
print("Hora formateada:", hora_formato)
```

Explicación Minuciosa: Se emplea `strftime("%H:%M:%S")` para formatear la hora. Los códigos **`%H`, `%M`, y `%S`** se utilizan para obtener la hora en formato de 24 horas, los minutos y los segundos, respectivamente. **Comandos y Controles Usados:**

- `strftime()`: Formatea la hora.
- `%H`: Hora en formato 24h.
- `%M`: Minutos.
- `%S`: Segundos.

Ejercicio 6

Enunciado: Calcula la diferencia de días entre dos fechas ingresadas por el usuario. **Solución:**

```
import datetime
fecha1 = input("Ingresa la primera fecha (dd/mm/yyyy): ")
fecha2 = input("Ingresa la segunda fecha (dd/mm/yyyy): ")
f1 = datetime.datetime.strptime(fecha1, "%d/%m/%Y")
f2 = datetime.datetime.strptime(fecha2, "%d/%m/%Y")
diferencia = f2 - f1
print("Diferencia en días:", diferencia.days)
```

Explicación Minuciosa: Primero, se recibe la información del usuario mediante `input()`. Luego, la función **`strptime()`** es fundamental, ya que convierte la cadena de texto ingresada por el usuario en un objeto `datetime` para que pueda ser operado. Al restar los dos objetos `datetime`

(`f2 - f1`), se obtiene un objeto `timedelta`, del cual se extrae la cantidad de días utilizando `.days`. **Comandos y Controles Usados:**

- `input()`: Recibe información del usuario.
- `datetime.datetime.strptime(cadena, formato)`: Convierte la cadena de texto a un objeto fecha.
- `.days`: Devuelve la cantidad de días de la diferencia de tiempo (`timedelta`).

Ejercicio 7

Enunciado: Muestra el día de la semana de la fecha actual. **Solución:**

```
import datetime
hoy = datetime.datetime.now()
dias = ["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"]
print("Hoy es:", dias[hoy.weekday()])
```

Explicación Minuciosa: La clave de este ejercicio es el método `.weekday()`, que se aplica al objeto `datetime` y devuelve un número entero que va de 0 (Lunes) a 6 (Domingo). Se utiliza una lista (`dias`) para traducir ese número a su respectivo nombre del día de la semana. **Comandos y Controles Usados:**

- `datetime.datetime.now()`: Obtiene la fecha actual.
- `.weekday()`: Devuelve el número del día de la semana (0=Lunes, 6=Domingo).

Ejercicio 8

Enunciado: Suma 10 días a la fecha actual y muestra el resultado. **Solución:**

```
import datetime
hoy = datetime.datetime.now()
futuro = hoy + datetime.timedelta(days=10)
print("Fecha dentro de 10 días:", futuro)
```

Explicación Minuciosa: Se utiliza la clase `datetime.timedelta` para representar un intervalo de tiempo. Al definir `timedelta(days=10)`, se crea un objeto que se puede sumar a la fecha actual (`hoy + ...`) para calcular una fecha futura. **Comandos y Controles Usados:**

- `datetime.timedelta(days=n)`: Representa un intervalo de tiempo (en este caso, 10 días).
- `+`: Suma el intervalo de tiempo (`timedelta`) a la fecha.

Ejercicio 9

Enunciado: Resta 7 días a una fecha ingresada por el usuario. **Solución:**

```
import datetime
fecha = input("Ingresa una fecha (dd/mm/yyyy): ")
f = datetime.datetime.strptime(fecha, "%d/%m/%Y")
resultado = f - datetime.timedelta(days=7)
print("Fecha 7 días antes:", resultado)
```

Explicación Minuciosa: El mecanismo es similar al Ejercicio 8, pero se utiliza el operador de resta (`-`) para sustraer un intervalo de tiempo (`timedelta(days=7)`) de la fecha ya convertida (usando `strptime()`). **Comandos y Controles Usados:**

- `- datetime.timedelta(days=7)`: Resta un intervalo de 7 días.
- `.strptime()`: Convierte la cadena ingresada por el usuario a un objeto fecha.

Ejercicio 10

Enunciado: Calcula cuántos segundos han pasado desde la medianoche de hoy. **Solución:**

```
import datetime
ahora = datetime.datetime.now()
medianoche = datetime.datetime.combine(datetime.date.today(),
datetime.time(0,0,0))
segundos = (ahora - medianoche).total_seconds()
print("Segundos desde medianoche:", segundos)
```

Explicación Minuciosa: Para determinar la medianoche de hoy, se utiliza **`datetime.combine()`**, que une la fecha actual (`datetime.date.today()`) con una hora específica (00:00:00, creada con `datetime.time(0,0,0)`). Al restar la hora actual (`ahora`) de la medianoche, se obtiene un `timedelta`. El método **`.total_seconds()`** convierte ese `timedelta` en la cantidad total de segundos transcurridos. **Comandos y Controles Usados:**

- `datetime.date.today()`: Obtiene la fecha de hoy.
- `datetime.time(0,0,0)`: Define la hora 00:00:00.
- `(ahora - medianoche).total_seconds()`: Calcula los segundos transcurridos en la diferencia.

Ejercicio 11

Enunciado: Convierte una fecha ingresada por el usuario a formato yyyy-mm-dd. **Solución:**

```
import datetime
fecha = input("Ingresa una fecha (dd/mm/yyyy): ")
f = datetime.datetime.strptime(fecha, "%d/%m/%Y")
print("Fecha en formato ISO:", f.strftime("%Y-%m-%d"))
```

Explicación Minuciosa: Después de convertir la cadena de entrada a un objeto fecha usando `strptime()`, se usa **`strftime()`** con los códigos `%Y-%m-%d` para transformar la fecha al formato ISO 8601 (año-mes-día). **Comandos y Controles Usados:**

- `strptime()`: Convierte cadena a objeto fecha.
- `strftime("%Y-%m-%d")`: Formatea la fecha a año-mes-día.

Ejercicio 12

Enunciado: Imprime la fecha actual con nombre de mes y día de la semana. **Solución:**

```
import datetime
hoy = datetime.datetime.now()
print(hoy.strftime("%A, %d de %B de %Y"))
```

Explicación Minuciosa: Este ejercicio requiere el uso de códigos de formato que representan nombres completos. **`%A`** se utiliza para el nombre del día de la semana y **`%B`** para el nombre completo del mes. **Comandos y Controles Usados:**

- `%A`: Nombre del día de la semana.
- `%B`: Nombre del mes.
- `%d` y `%Y`: Día y año.

Ejercicio 13

Enunciado: Calcula los días restantes hasta fin de año. **Solución:**

```
import datetime
hoy = datetime.date.today()
fin_año = datetime.date(hoy.year, 12, 31)
faltan = (fin_año - hoy).days
print("Días hasta fin de año:", faltan)
```

Explicación Minuciosa: Se crea la fecha específica del 31 de diciembre del año actual usando **`datetime.date(año, mes, día)`**. Luego, se resta la fecha actual a la fecha de fin de año y se extraen los días de la diferencia con `.days`. **Comandos y Controles Usados:**

- `datetime.date(año, mes, día)`: Crea una fecha específica.
- `.days`: Obtiene los días de la diferencia.

Ejercicio 14

Enunciado: Calcula cuántos días viviste según tu fecha de nacimiento. **Solución:**

```
import datetime
nacimiento = input("Ingresa tu fecha de nacimiento (dd/mm/yyyy): ")
f_nac = datetime.datetime.strptime(nacimiento, "%d/%m/%Y")
hoy = datetime.datetime.now()
dias_vividos = (hoy - f_nac).days
print("Has vivido", dias_vividos, "días")
```

Explicación Minuciosa: El programa convierte la fecha de nacimiento ingresada a un objeto `datetime` y luego resta esa fecha de la fecha actual (`hoy - f_nac`). El resultado es un `timedelta`, del cual se extrae directamente la cantidad de días vividos usando **`.days`**. **Comandos y Controles Usados:**

- `(hoy - f_nac).days`: Calcula la diferencia en días entre las dos fechas.

Ejercicio 15

Enunciado: Muestra la fecha y hora actuales en formato de 12 horas con AM/PM. **Solución:**

```
import datetime
ahora = datetime.datetime.now()
print("Hora actual:", ahora.strftime("%I:%M:%S %p"))
```

Explicación Minuciosa: Para obtener el formato de 12 horas, se utiliza el código **`%I`** para la hora. Además, se utiliza el código **`%p`** para añadir el indicador AM o PM. **Comandos y Controles Usados:**

- `%I`: Hora en formato 12h.
- `%p`: Indicador AM/PM.

Ejercicio 16

Enunciado: Calcula cuántos días faltan para tu próximo cumpleaños. **Solución:**

```
import datetime
cumple = input("Ingresa tu cumpleaños (dd/mm): ")
dia, mes = map(int, cumple.split("/"))
hoy = datetime.date.today()
proximo = datetime.date(hoy.year, mes, dia)
if proximo < hoy:
proximo = datetime.date(hoy.year + 1, mes, dia)
faltan = (proximo - hoy).days
print("Días para tu próximo cumpleaños:", faltan)
```

Explicación Minuciosa: El usuario ingresa el día y mes, que se separan usando **`split("/")`** y se convierten a enteros con **`map(int, ...)`**. La lógica crucial es **determinar si el cumpleaños ya pasó este año** (`if proximo < hoy:`). Si ya pasó, la fecha del próximo cumpleaños se ajusta al año siguiente (`hoy.year + 1`). Finalmente, la resta de fechas da el número de días faltantes. **Comandos y Controles Usados:**

- `split("/")`: Separa el día y el mes de la cadena.
- `map(int, ...)`: Convierte las partes de la fecha a números enteros.
- `if proximo < hoy`: Control de flujo que ajusta la fecha al siguiente año si el cumpleaños ya ha ocurrido.

Ejercicio 17

Enunciado: Calcula cuántos segundos faltan para la medianoche. **Solución:**

```
import datetime
ahora = datetime.datetime.now()
medianoche = datetime.datetime.combine(datetime.date.today() +
datetime.timedelta(days=1), datetime.time(0,0,0))
faltan_segundos = (medianoche - ahora).total_seconds()
print("Segundos hasta medianoche:", int(faltan_segundos))
```

Explicación Minuciosa: Para calcular el tiempo restante, se debe determinar primero la medianoche del día siguiente. Esto se logra sumando un día a la fecha actual con **`datetime.timedelta(days=1)`** y luego combinándolo con la hora 00:00:00. La diferencia entre esta medianoche futura y la hora actual se convierte en segundos enteros usando **`.total_seconds()`**. **Comandos y Controles Usados:**

- `datetime.timedelta(days=1)`: Suma un día (se usa para obtener la fecha de mañana).
- `.total_seconds()`: Convierte el `timedelta` resultante a segundos.

Ejercicio 18

Enunciado: Muestra la fecha y hora actuales con microsegundos. **Solución:**

```
import datetime
ahora = datetime.datetime.now()
print("Fecha y hora con microsegundos:", ahora.strftime("%Y-%m-%d %H:%M:%S.%f"))
```

Explicación Minuciosa: Para incluir la precisión de microsegundos en la representación de la hora, se utiliza el código de formato **`%f`** dentro del método `strftime()`. Este código muestra 6 dígitos para los microsegundos. **Comandos y Controles Usados:**

- `%f`: Representa los microsegundos (6 dígitos).

Ejercicio 19

Enunciado: Imprime la fecha actual y el número de semana del año. **Solución:**

```
import datetime
hoy = datetime.date.today()
print("Semana del año:", hoy.isocalendar())
```

Explicación Minuciosa: El método **`isocalendar()`** se utiliza para obtener información basada en la norma ISO 8601. Este método devuelve una tupla que contiene el año ISO, el número de semana ISO y el día de la semana ISO. Al acceder al índice ``, se extrae el número de semana requerido. **Comandos y Controles Usados:**

- `isocalendar()`: Extrae específicamente el número de semana del año.

Ejercicio 20

Enunciado: Calcula la diferencia en años, meses y días entre dos fechas. **Solución:**

```
import datetime
from dateutil.relativedelta import relativedelta
f1 = datetime.datetime.strptime(input("Fecha inicial (dd/mm/yyyy): "),
"%d/%m/%Y")
f2 = datetime.datetime.strptime(input("Fecha final (dd/mm/yyyy): "), "%d/%m/%Y")
diff = relativedelta(f2, f1)
print(f"Diferencia: {diff.years} años, {diff.months} meses, {diff.days} días")
```

Explicación Minuciosa: Este ejercicio requiere la importación de **`relativedelta`** del módulo `dateutil`. A diferencia de la simple resta de objetos `datetime`, `relativedelta(f2, f1)` calcula la diferencia exacta en términos de años, meses y días, permitiendo acceder a estos atributos (`.years`, `.months`, `.days`) de manera precisa. **Comandos y Controles Usados:**

- `from dateutil.relativedelta import relativedelta`: Permite realizar cálculos precisos entre fechas que incluyen años y meses.
- `relativedelta(f2, f1)`: Calcula la diferencia en años, meses y días.

Ejercicio 21

Enunciado: Calcula cuántos días hábiles hay entre dos fechas (excluyendo fines de semana).
Solución:

```
import datetime
fecha_inicio = datetime.datetime.strptime(input("Fecha inicio (dd/mm/yyyy): "),
"%d/%m/%Y")
fecha_fin = datetime.datetime.strptime(input("Fecha fin (dd/mm/yyyy): "),
"%d/%m/%Y")
dias_habiles = 0
fecha_actual = fecha_inicio
while fecha_actual <= fecha_fin:
if fecha_actual.weekday() < 5:  # 0=lunes, 6=domingo
dias_habiles += 1
fecha_actual += datetime.timedelta(days=1)
print("Días hábiles:", dias_habiles)
```

Explicación Minuciosa: La solución implementa un **bucle `while`** para recorrer día por día desde la fecha de inicio hasta la fecha de fin. Dentro del bucle, se utiliza el método **`weekday()`** para comprobar el día de la semana. Un día hábil se identifica si `weekday()` es menor que 5 (Lunes=0, Martes=1... Viernes=4). En cada iteración, se suma un día usando `datetime.timedelta(days=1)`. **Comandos y Controles Usados:**

- `weekday() < 5`: Comprueba que la fecha no sea sábado (5) ni domingo (6).
- `datetime.timedelta(days=1)`: Suma un día para avanzar la fecha en el bucle.

Ejercicio 22

Enunciado: Convierte segundos a horas, minutos y segundos. **Solución:**

```
segundos_totales = int(input("Ingresa los segundos: "))
horas = segundos_totales // 3600
minutos = (segundos_totales % 3600) // 60
segundos = segundos_totales % 60
print(f"{horas} horas, {minutos} minutos, {segundos} segundos")
```

Explicación Minuciosa: Para la conversión, se utilizan operadores aritméticos. Las **horas** se obtienen mediante la división entera (`//`) de los segundos totales por 3600 (segundos en una hora). Los **minutos** y los **segundos** se obtienen utilizando el operador **módulo (%)** para calcular el residuo, seguido de división entera cuando corresponde. **Comandos y Controles Usados:**

- `//`: División entera, utilizada para calcular horas y minutos completos.
- %: Operador módulo, utilizado para obtener el residuo de la división (por ejemplo, los segundos restantes después de extraer las horas completas).

Ejercicio 23

Enunciado: Imprime el calendario del mes actual. **Solución:**

```
import calendar
import datetime
hoy = datetime.date.today()
print(calendar.month(hoy.year, hoy.month))
```

Explicación Minuciosa: Este ejercicio requiere importar el módulo **`calendar`**. La función **`calendar.month(año, mes)`** toma el año y el mes de la fecha actual (`hoy.year`, `hoy.month`) y genera una representación formateada del calendario de ese mes. **Comandos y Controles Usados:**

- `import calendar`: Importa el módulo para la manipulación y visualización de calendarios.
- `calendar.month(año, mes)`: Imprime el mes en un formato de calendario.

Ejercicio 24

Enunciado: Calcula la fecha y hora exacta dentro de 1000000 segundos desde ahora. **Solución:**

```
import datetime
ahora = datetime.datetime.now()
futuro = ahora + datetime.timedelta(seconds=1000000)
print("Fecha y hora dentro de 1,000,000 segundos:", futuro)
```

Explicación Minuciosa: Similar al Ejercicio 8, se usa **`datetime.timedelta`**, pero en este caso se especifica el intervalo en segundos (`seconds=1000000`). Este intervalo se suma al objeto `datetime` actual. **Comandos y Controles Usados:**

- `timedelta(seconds=n)`: Representa un intervalo de n segundos que puede ser sumado a una fecha/hora.

Ejercicio 25

Enunciado: Muestra la fecha actual en formato "Miércoles, 15 de Enero de 2025, 14:35:20". **Solución:**

```
import datetime
ahora = datetime.datetime.now()
print(ahora.strftime("%A, %d de %B de %Y, %H:%M:%S"))
```

Explicación Minuciosa: Se combinan códigos de formato para obtener la fecha y hora completas. Se utilizan **`%A`** y **`%B`** para los nombres completos del día de la semana y del mes, respectivamente, y `%H:%M:%S` para la hora en formato 24 horas. **Comandos y Controles Usados:**

- `%A`: Día de la semana completo.
- `%B`: Mes completo (en palabras).
- `%H:%M:%S`: Hora en formato 24h, minutos y segundos.

Ejercicio 26

Enunciado: Calcula el número de días entre dos fechas ignorando los años bisiestos. **Solución:**

```
import datetime
f1 = datetime.datetime.strptime(input("Fecha 1 (dd/mm/yyyy): "), "%d/%m/%Y")
f2 = datetime.datetime.strptime(input("Fecha 2 (dd/mm/yyyy): "), "%d/%m/%Y")
dias = (f2 - f1).days
print("Días entre fechas (ignorando bisiestos):", dias)
```

Explicación Minuciosa: La resta simple de dos objetos `datetime` (`f2 - f1`) devuelve un `timedelta`. Al obtener `.days` de ese `timedelta`, Python calcula la diferencia exacta en días, manejando automáticamente la consideración de los años bisiestos. **Comandos y Controles Usados:**

- `(f2 - f1).days`: Resta las fechas y obtiene el número total de días de diferencia.

Ejercicio 27

Enunciado: Calcula cuántos viernes hay entre dos fechas. **Solución:**

```
import datetime
fecha_inicio = datetime.datetime.strptime(input("Fecha inicio (dd/mm/yyyy): "),
"%d/%m/%Y")
fecha_fin = datetime.datetime.strptime(input("Fecha fin (dd/mm/yyyy): "),
"%d/%m/%Y")
viernes = 0
fecha_actual = fecha_inicio
while fecha_actual <= fecha_fin:
if fecha_actual.weekday() == 4:  # Viernes
viernes += 1
fecha_actual += datetime.timedelta(days=1)
print("Cantidad de viernes:", viernes)
```

Explicación Minuciosa: Similar al Ejercicio 21, se utiliza un bucle para iterar día a día. En este caso, el control clave es la condición `if fecha_actual.weekday() == 4`, que **identifica específicamente el viernes** (donde 4 es el índice para viernes). Se incrementa un contador si se cumple la condición. **Comandos y Controles Usados:**

- `weekday() == 4`: Comprueba si el día actual es viernes.

Ejercicio 28

Enunciado: Calcula la edad exacta en años, meses y días a partir de la fecha de nacimiento. **Solución:**

```
import datetime
from dateutil.relativedelta import relativedelta
nacimiento = datetime.datetime.strptime(input("Fecha de nacimiento (dd/mm/yyyy):
"), "%d/%m/%Y")
hoy = datetime.datetime.now()
edad = relativedelta(hoy, nacimiento)
print(f"Tienes {edad.years} años, {edad.months} meses y {edad.days} días")
```

Explicación Minuciosa: Este cálculo preciso de la edad requiere el uso de **`relativedelta`** (importado del módulo `dateutil`). `relativedelta` permite obtener la diferencia exacta de tiempo entre la fecha actual (`hoy`) y el nacimiento, devolviendo años, meses y días como atributos accesibles. **Comandos y Controles Usados:**

- `relativedelta(f2, f1)`: Proporciona la diferencia precisa de tiempo en unidades de años, meses y días.

Ejercicio 29

Enunciado: Imprime la hora actual en distintos husos horarios (UTC y local). **Solución:**

```
from datetime import datetime, timezone
hora_utc = datetime.now(timezone.utc)
hora_local = datetime.now()
print("Hora UTC:", hora_utc)
print("Hora local:", hora_local)
```

Explicación Minuciosa: Para obtener la Hora Universal Coordinada (UTC), se usa `datetime.now()` pasando como argumento **`timezone.utc`**. La hora local se obtiene mediante `datetime.now()` sin argumentos de zona horaria. **Comandos y Controles Usados:**

- `timezone.utc`: Define la zona horaria UTC.
- `datetime.now(timezone)`: Obtiene la hora según la zona horaria especificada.

Ejercicio 30

Enunciado: Convierte un timestamp (segundos desde 1970) a fecha legible. **Solución:**

```
import datetime
timestamp = int(input("Ingresa un timestamp: "))
fecha = datetime.datetime.fromtimestamp(timestamp)
print("Fecha correspondiente:", fecha)
```

Explicación Minuciosa: Un timestamp representa los segundos transcurridos desde el "epoch" (1 de enero de 1970). La función **`datetime.datetime.fromtimestamp(n)`** convierte ese número de segundos (`n`) en un objeto `datetime` legible. **Comandos y Controles Usados:**

- `fromtimestamp(n)`: Transforma los segundos transcurridos desde 1970 a una fecha y hora.

Ejercicio 31

Enunciado: Calcula cuántos días faltan para el próximo viernes 13. **Solución:**

```
import datetime
hoy = datetime.date.today()
año = hoy.year
while True:
for mes in range(1, 13):
fecha = datetime.date(año, mes, 13)
if fecha.weekday() == 4 and fecha > hoy:
print("El próximo viernes 13 es:", fecha)
break
else:
año += 1
continue
break
```

Explicación Minuciosa: Este programa utiliza un bucle anidado que incrementa el año y recorre los meses (1 al 12). En cada iteración, se construye la fecha del día 13 del mes. Se verifica que: 1) el día de la semana sea viernes (`weekday() == 4`), y 2) que la fecha sea posterior al día de hoy

(`fecha > hoy`). Si se encuentra, el bucle se detiene; de lo contrario, se incrementa el año.
Comandos y Controles Usados:

- `weekday() == 4`: Condición para identificar que un día es viernes.
- `datetime.date(año, mes, día)`: Crea una fecha específica para el día 13.

Ejercicio 32

Enunciado: Calcula la duración de un evento ingresando fecha y hora de inicio y fin. **Solución:**

```
import datetime
inicio = datetime.datetime.strptime(input("Inicio (dd/mm/yyyy HH:MM): "),
"%d/%m/%Y %H:%M")
fin = datetime.datetime.strptime(input("Fin (dd/mm/yyyy HH:MM): "), "%d/%m/%Y
%H:%M")
duracion = fin - inicio
print("Duración del evento:", duracion)
```

Explicación Minuciosa: Se utiliza **`datetime.strptime()`** con el formato extendido (`%d/%m/%Y %H:%M`) para convertir las cadenas de entrada en objetos `datetime` que incluyen tanto fecha como hora. La duración se calcula mediante la resta directa (`fin - inicio`), que resulta en un objeto `timedelta` que muestra la duración. **Comandos y Controles Usados:**

- `datetime.strptime(cadena, formato)`: Convierte la cadena a un objeto fecha y hora.
- `fin - inicio`: Resta de dos fechas para obtener la duración (`timedelta`).

Ejercicio 33

Enunciado: Calcula el número de semanas completas entre dos fechas. **Solución:**

```
import datetime
f1 = datetime.datetime.strptime(input("Fecha 1 (dd/mm/yyyy): "), "%d/%m/%Y")
f2 = datetime.datetime.strptime(input("Fecha 2 (dd/mm/yyyy): "), "%d/%m/%Y")
semanas = (f2 - f1).days // 7
print("Semanas completas entre fechas:", semanas)
```

Explicación Minuciosa: Se calcula la diferencia total de días (`.days`) entre las dos fechas. Luego, esa cantidad de días se divide por 7 usando el operador de **división entera (`//`)**. Esto asegura que solo se cuenten las semanas que están completas (descartando fracciones). **Comandos y Controles Usados:**

- `(f2 - f1).days // 7`: Calcula las semanas completas usando la división entera de la diferencia de días.

Ejercicio 34

Enunciado: Convierte fecha y hora locales a UTC. **Solución:**

```
from datetime import datetime, timezone
local = datetime.strptime(input("Fecha y hora local (dd/mm/yyyy HH:MM): "),
"%d/%m/%Y %H:%M")
local_utc = local.astimezone(timezone.utc)
print("Hora en UTC:", local_utc)
```

Explicación Minuciosa: La función **`.astimezone(timezone.utc)`** toma un objeto `datetime` local y lo convierte a la zona horaria UTC. Esto es esencial para estandarizar la hora independientemente de la ubicación geográfica. **Comandos y Controles Usados:**

- `astimezone(timezone.utc)`: Convierte la hora local al estándar UTC.

Ejercicio 35

Enunciado: Calcula el tiempo transcurrido desde el inicio del año hasta hoy. **Solución:**

```
import datetime
hoy = datetime.date.today()
inicio_año = datetime.date(hoy.year, 1, 1)
dias_transcurridos = (hoy - inicio_año).days
print("Días transcurridos desde el inicio del año:", dias_transcurridos)
```

Explicación Minuciosa: Se crea la fecha específica del 1 de enero del año actual utilizando **`datetime.date(hoy.year, 1, 1)`**. Al restar esta fecha del inicio del año a la fecha de hoy, se obtiene el número total de días transcurridos. **Comandos y Controles Usados:**

- `datetime.date(año, mes, día)`: Crea la fecha de inicio del año (1 de enero).

Ejercicio 36

Enunciado: Calcula la cantidad de horas trabajadas en una semana ingresando horas de entrada y salida por día. **Solución:**

```
import datetime
total_horas = 0
for i in range(5):
entrada = input(f"Entrada día {i+1} (HH:MM): ")
salida = input(f"Salida día {i+1} (HH:MM): ")
f_entrada = datetime.datetime.strptime(entrada, "%H:%M")
f_salida = datetime.datetime.strptime(salida, "%H:%M")
duracion = f_salida - f_entrada
total_horas += duracion.seconds / 3600
print("Horas trabajadas en la semana:", total_horas)
```

Explicación Minuciosa: Se utiliza un bucle **`for`** para solicitar las horas de entrada y salida durante 5 días. La duración diaria se calcula restando las horas de entrada y salida (resultando en un `timedelta`). Para acumular el total en horas, se extraen los segundos del `timedelta` (`duracion.seconds`) y se dividen por 3600 (segundos por hora). **Comandos y Controles Usados:**

- `duracion.seconds / 3600`: Convierte la duración de segundos a horas y la suma al total.

Ejercicio 37

Enunciado: Imprime la fecha y hora actual en formato ISO 8601. **Solución:**

```
import datetime
ahora = datetime.datetime.now()
print("Fecha y hora ISO 8601:", ahora.isoformat())
```

Explicación Minuciosa: El método **`.isoformat()`** es la manera más sencilla de obtener la fecha y hora en el estándar internacional ISO 8601, sin necesidad de especificar códigos de formato complejos. **Comandos y Controles Usados:**

- `isoformat()`: Convierte el objeto `datetime` al formato estándar ISO 8601.

Ejercicio 38

Enunciado: Calcula la fecha del próximo lunes a partir de hoy. **Solución:**

```
import datetime
hoy = datetime.date.today()
dias_a_lunes = (0 - hoy.weekday() + 7) % 7
if dias_a_lunes == 0:
dias_a_lunes = 7
proximo_lunes = hoy + datetime.timedelta(days=dias_a_lunes)
print("Próximo lunes:", proximo_lunes)
```

Explicación Minuciosa: Se utiliza `weekday()` (donde 0 es lunes). La expresión `(0 - hoy.weekday() + 7) % 7` calcula la diferencia de días hasta el próximo lunes. El uso de **`% 7`** (módulo 7) asegura que el resultado esté en el rango de 0 a 6. Si el resultado es 0, significa que hoy es lunes, por lo que el programa se ajusta para mostrar el lunes de la semana siguiente (7 días). **Comandos y Controles Usados:**

- `weekday()`: Devuelve el número del día de la semana (Lunes=0).
- `% 7`: Operador módulo que asegura que el número de días a sumar esté en el rango correcto.

Ejercicio 39

Enunciado: Calcula cuántos meses completos han pasado desde una fecha dada. **Solución:**

```
from dateutil.relativedelta import relativedelta
import datetime
fecha = datetime.datetime.strptime(input("Ingresa fecha (dd/mm/yyyy): "),
"%d/%m/%Y")
hoy = datetime.datetime.now()
diferencia = relativedelta(hoy, fecha)
meses = diferencia.years * 12 + diferencia.months
print("Meses completos transcurridos:", meses)
```

Explicación Minuciosa: Al igual que en los ejercicios de edad, **`relativedelta`** es esencial para obtener la diferencia precisa en años y meses. La cantidad total de meses completos se calcula multiplicando el atributo de años (`diferencia.years`) por 12 y sumando los meses restantes (`diferencia.months`). **Comandos y Controles Usados:**

- `relativedelta(f2, f1)`: Proporciona la diferencia precisa en años, meses y días.

Ejercicio 40

Enunciado: Muestra la fecha y hora actuales en tres husos horarios distintos. **Solución:**

```
from datetime import datetime, timezone, timedelta
ahora_utc = datetime.now(timezone.utc)
hora_nueva_york = ahora_utc + timedelta(hours=-5)
```

```
hora_tokio = ahora_utc + timedelta(hours=9)
hora_londres = ahora_utc + timedelta(hours=0)
print("Hora UTC:", ahora_utc)
print("Hora Nueva York:", hora_nueva_york)
print("Hora Tokio:", hora_tokio)
print("Hora Londres:", hora_londres)
```

Explicación Minuciosa: Se toma la Hora UTC como base, obtenida usando `timezone.utc`. Luego, se utiliza **`timedelta(hours=n)`** para sumar o restar las horas necesarias para ajustar al huso horario deseado (e.g., -5 para Nueva York, +9 para Tokio, 0 para Londres/UTC). **Comandos y Controles Usados:**

- `timedelta(hours=n)`: Se usa para sumar o restar horas a un objeto `datetime`.
- `timezone.utc`: Establece la base de la hora universal.

Ejercicio 41

Enunciado: Crea un programa que calcule cuántos días faltan para el próximo año bisiesto.
Solución:

```
import datetime
hoy = datetime.date.today()
año = hoy.year + 1
while True:
if (año % 4 == 0 and año % 100 != 0) or (año % 400 == 0):
bisiesto = datetime.date(año, 2, 29)
dias_faltantes = (bisiesto - hoy).days
print("Días hasta el próximo año bisiesto:", dias_faltantes)
break
año += 1
```

Explicación Minuciosa: El programa inicia buscando desde el año siguiente al actual (`hoy.year + 1`). Utiliza un bucle **`while True`** y la **regla matemática para determinar si un año es bisiesto** (`(año % 4 == 0 and año % 100 != 0) or (año % 400 == 0)`). Una vez que se encuentra el año bisiesto, se calcula la fecha del 29 de febrero y se resta a la fecha actual para obtener los días faltantes. **Comandos y Controles Usados:**

- `datetime.date(año, mes, día)`: Crea una fecha específica (en este caso, 29 de febrero).
- `(año % 4 == 0 and año % 100 != 0) or (año % 400 == 0)`: La lógica de control para identificar un año bisiesto.

Ejercicio 42

Enunciado: Calcula el tiempo transcurrido desde una fecha y hora específica hasta ahora en días, horas y minutos. **Solución:**

```
import datetime
fecha_hora = datetime.datetime.strptime(input("Fecha y hora (dd/mm/yyyy HH:MM):
"), "%d/%m/%Y %H:%M")
ahora = datetime.datetime.now()
diferencia = ahora - fecha_hora
dias = diferencia.days
horas = diferencia.seconds // 3600
minutos = (diferencia.seconds % 3600) // 60
```

```
print(f"Tiempo transcurrido: {dias} días, {horas} horas, {minutos} minutos")
```

Explicación Minuciosa: La diferencia entre las dos fechas (`diferencia`) es un objeto `timedelta`. Este objeto se descompone en días usando el atributo **`.days`**. Las horas y minutos restantes se calculan a partir del atributo **`.seconds`** del `timedelta`, usando división entera (`//`) y módulo (%) de manera similar al Ejercicio 22. **Comandos y Controles Usados:**

- `diferencia.seconds // 3600`: Obtiene las horas completas del tiempo restante.
- `(diferencia.seconds % 3600) // 60`: Obtiene los minutos restantes.

Ejercicio 43

Enunciado: Crea un cronómetro que muestre segundos transcurridos hasta que el usuario presione Enter. **Solución:**

```
import time
input("Presiona Enter para iniciar el cronómetro...")
inicio = time.time()
input("Presiona Enter para detener el cronómetro...")
fin = time.time()
print("Segundos transcurridos:", fin - inicio)
```

Explicación Minuciosa: Este ejercicio utiliza el módulo **`time`**. La función **`time.time()`** devuelve la cantidad de segundos transcurridos desde el epoch (1970). Al restar el valor de `time.time()` al inicio del cronómetro del valor al final (`fin - inicio`), se obtiene el tiempo transcurrido en segundos. **Comandos y Controles Usados:**

- `import time`: Importa el módulo para funciones relacionadas con el tiempo.
- `time.time()`: Devuelve los segundos transcurridos desde el epoch.

Ejercicio 44

Enunciado: Calcula el número de días que tiene cada mes del año actual. **Solución:**

```
import calendar
año = int(input("Ingresa el año: "))
for mes in range(1, 13):
dias = calendar.monthrange(año, mes)
print(f"Mes {mes} tiene {dias} días")
```

Explicación Minuciosa: Se utiliza un bucle `for` para iterar a través de los 12 meses. La función clave es **`calendar.monthrange(año, mes)`**, que devuelve una tupla. El primer elemento es el día de la semana del primer día, y el segundo elemento (``) es la cantidad total de días que tiene ese mes. **Comandos y Controles Usados:**

- `calendar.monthrange(año, mes)`: Obtiene el número de días que tiene el mes especificado.

Ejercicio 45

Enunciado: Muestra la fecha actual en formato europeo, americano y ISO. **Solución:**

```
import datetime
```

```
hoy = datetime.datetime.now()
formato_europeo = hoy.strftime("%d/%m/%Y")
formato_americano = hoy.strftime("%m/%d/%Y")
formato_iso = hoy.isoformat()
print("Formato europeo:", formato_europeo)
print("Formato americano:", formato_americano)
print("Formato ISO:", formato_iso)
```

Explicación Minuciosa: Se utilizan combinaciones de `strftime()` para el formato europeo (día/mes/año, `%d/%m/%Y`) y el formato americano (mes/día/año, `%m/%d/%Y`). Para el formato ISO, se utiliza el método **`.isoformat()`**, que genera el estándar internacional. **Comandos y Controles Usados:**

- `%d/%m/%Y`: Representación de formato europeo.
- `%m/%d/%Y`: Representación de formato americano.
- `isoformat()`: Genera la representación estándar internacional.

Ejercicio 46

Enunciado: Calcula el día de la semana para cualquier fecha ingresada. **Solución:**

```
import datetime
fecha = datetime.datetime.strptime(input("Ingresa fecha (dd/mm/yyyy): "),
"%d/%m/%Y")
dias = ["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"]
print("Día de la semana:", dias[fecha.weekday()])
```

Explicación Minuciosa: Una vez que la fecha se ha convertido de cadena a objeto `datetime` con `strptime()`, se utiliza el método **`.weekday()`**. Este método devuelve un número entre 0 y 6. Este número se utiliza como índice para acceder al nombre del día correspondiente en una lista predefinida (`dias`). **Comandos y Controles Usados:**

- `weekday()`: Devuelve un valor numérico de 0 a 6 para los días de la semana (Lunes a Domingo).

Ejercicio 47

Enunciado: Crea un programa que calcule cuántos segundos hay en un año bisiesto y en uno normal. **Solución:**

```
segundos_normal = 365 * 24 * 3600
segundos_bisiesto = 366 * 24 * 3600
print("Segundos en un año normal:", segundos_normal)
print("Segundos en un año bisiesto:", segundos_bisiesto)
```

Explicación Minuciosa: El cálculo se basa en la multiplicación directa: días del año (365 o 366) por 24 horas por 3600 segundos/hora. **Comandos y Controles Usados:**

- `*`: Operador de multiplicación.
- `3600`: Número de segundos en una hora.

Ejercicio 48

Enunciado: Calcula la diferencia entre dos zonas horarias dadas (por ejemplo, Tokio y Nueva York). **Solución:**

```
from datetime import datetime, timedelta, timezone
tokio = datetime.now(timezone(timedelta(hours=9)))
nueva_york = datetime.now(timezone(timedelta(hours=-5)))
diferencia = tokio - nueva_york
print("Diferencia entre Tokio y Nueva York:", diferencia)
```

Explicación Minuciosa: Para definir una zona horaria específica (offset), se utiliza **`timezone(timedelta(hours=n))`**, donde `n` es el desfase horario respecto a UTC. Se obtiene la hora en Tokio (+9 horas) y la hora en Nueva York (-5 horas). La resta directa entre los dos objetos `datetime` zonificados calcula la diferencia horaria entre ellos. **Comandos y Controles Usados:**

- `timezone(timedelta(hours=n))`: Define una zona horaria con un offset específico.
- `datetime.now(timezone)`: Obtiene la hora actual para la zona horaria definida.

Ejercicio 49

Enunciado: Calcula la fecha de Pascua para un año dado usando el algoritmo de Meeus. **Solución:**

```
year = int(input("Ingresa el año: "))
a = year % 19
b = year // 100
c = year % 100
d = b // 4
e = b % 4
f = (b + 8) // 25
g = (b - f + 1) // 3
h = (19*a + b - d - g + 15) % 30
i = c // 4
k = c % 4
l = (32 + 2*e + 2*i - h - k) % 7
m = (a + 11*h + 22*l) // 451
mes = (h + l - 7*m + 114) // 31
dia = ((h + l - 7*m + 114) % 31) + 1
print(f"Pascua en {year}: {dia}/{mes}/{year}")
```

Explicación Minuciosa: Este programa implementa el complejo **Algoritmo de Meeus**, una serie de cálculos matemáticos para determinar la fecha de Pascua en el calendario gregoriano para cualquier año. La lógica se basa fundamentalmente en el uso extensivo de los operadores de **módulo (%)** y **división entera (`//`)** para seguir los pasos del algoritmo y derivar el mes y el día de Pascua. **Comandos y Controles Usados:**

- `%` y `//`: Módulo y división entera, utilizados para realizar los cálculos secuenciales requeridos por el algoritmo.

Ejercicio 50

Enunciado: Crea un programa que muestre un recordatorio en una fecha y hora futura específica. **Solución:**

```
import datetime
import time
recordatorio = datetime.datetime.strptime(input("Ingresa fecha y hora del
recordatorio (dd/mm/yyyy HH:MM): "), "%d/%m/%Y %H:%M")
print("Esperando el recordatorio...")
while datetime.datetime.now() < recordatorio:
time.sleep(10)
print("¡Es hora del recordatorio!")
```

Explicación Minuciosa: El programa convierte la entrada del usuario a un objeto `datetime` (fecha y hora del recordatorio). La ejecución se controla mediante un **bucle `while`** que se mantiene activo mientras la hora actual (`datetime.datetime.now()`) sea anterior a la hora del recordatorio. Dentro del bucle, **`time.sleep(10)`** pausa la ejecución por 10 segundos antes de verificar la hora nuevamente, optimizando el uso de recursos. **Comandos y Controles Usados:**

- `time.sleep(n)`: Pausa la ejecución del programa por n segundos (en este caso, 10 segundos).
- `while datetime.datetime.now() < recordatorio`: Bucle de control que espera hasta que se cumpla la hora de la alarma.

Ejercicio 1

Enunciado:
Imprime la fecha y hora actuales del sistema.

Solución:

```
import datetime


ahora = datetime.datetime.now()
print("Fecha y hora actuales:", ahora)
```

Explicación:

- `datetime.datetime.now()` obtiene la fecha y hora actuales del sistema.
- `print()` muestra el resultado en pantalla.

Comandos usados:

- `import datetime`: importa el módulo `datetime` para manejar fechas y horas.
- `datetime.datetime.now()`: obtiene la fecha y hora actuales.
- `print()`: imprime en pantalla.

Ejercicio 2

Enunciado:
Muestra solo la fecha actual.

Solución:

```
import datetime


fecha = datetime.date.today()
print("Fecha actual:", fecha)
```

Explicación:

- `datetime.date.today()` devuelve solo la fecha actual.

Comandos usados:

- `datetime.date.today()`: obtiene la fecha actual (año, mes, día).

Ejercicio 3

Enunciado:
Muestra solo la hora actual en formato de 24 horas.

Solución:

```
import datetime


hora = datetime.datetime.now().time()
print("Hora actual:", hora)
```

Explicación:

- `.time()` extrae solo la hora del objeto `datetime`.

Comandos usados:

- `datetime.datetime.now()`: obtiene la fecha y hora actuales.
- `.time()`: extrae la hora.
- `print()`: imprime en pantalla.

Ejercicio 4

Enunciado:
Imprime la fecha actual en formato `dd/mm/yyyy`.

Solución:

```
import datetime


ahora = datetime.datetime.now()
fecha_formato = ahora.strftime("%d/%m/%Y")
print("Fecha formateada:", fecha_formato)
```

Explicación:

- `strftime()` convierte la fecha a un formato legible según los códigos.

Comandos usados:

- `%d`: día con dos dígitos
- `%m`: mes con dos dígitos
- `%Y`: año con cuatro dígitos

Ejercicio 5

Enunciado:
Imprime la hora actual en formato `HH:MM:SS`.

Solución:

```
import datetime


ahora = datetime.datetime.now()
hora_formato = ahora.strftime("%H:%M:%S")
print("Hora formateada:", hora_formato)
```

Explicación:

- `strftime("%H:%M:%S")` convierte la hora en formato legible de 24 horas.

Comandos usados:

- `%H`: hora en formato 24h
- `%M`: minutos
- `%S`: segundos

Ejercicio 6

Enunciado:
Calcula la diferencia de días entre dos fechas ingresadas por el usuario.

Solución:

```
import datetime

fecha1 = input("Ingresa la primera fecha (dd/mm/yyyy): ")
fecha2 = input("Ingresa la segunda fecha (dd/mm/yyyy): ")

f1 = datetime.datetime.strptime(fecha1, "%d/%m/%Y")
f2 = datetime.datetime.strptime(fecha2, "%d/%m/%Y")

diferencia = f2 - f1
print("Diferencia en días:", diferencia.days)
```

Explicación:

- `strptime()` convierte un string a objeto `datetime`.
- Restando dos fechas se obtiene un `timedelta` que tiene `.days`.

Comandos usados:

- `input()`: recibe información del usuario.
- `datetime.datetime.strptime(cadena, formato)`: convierte cadena a fecha.
- `.days`: devuelve la cantidad de días de la diferencia.

Ejercicio 7

Enunciado:
Muestra el día de la semana de la fecha actual.

Solución:

```
import datetime


hoy = datetime.datetime.now()
dias = ["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"]
print("Hoy es:", dias[hoy.weekday()])
```

Explicación:

- `weekday()` devuelve un número de 0 a 6 según el día de la semana.
- La lista `dias` traduce ese número a nombre.

Comandos usados:

- `datetime.datetime.now()`: fecha actual
- `.weekday()`: número del día de la semana
- `print()`: muestra el resultado

Ejercicio 8

Enunciado:
Suma 10 días a la fecha actual y muestra el resultado.

Solución:

```
import datetime


hoy = datetime.datetime.now()
futuro = hoy + datetime.timedelta(days=10)
print("Fecha dentro de 10 días:", futuro)
```

Explicación:

- `timedelta(days=10)` permite sumar o restar días.

Comandos usados:

- `datetime.timedelta(days=n)`: representa un intervalo de tiempo
- `+`: suma el intervalo a la fecha
- `print()`: imprime el resultado

Ejercicio 9

Enunciado:
Resta 7 días a una fecha ingresada por el usuario.

Solución:

```
import datetime


fecha = input("Ingresa una fecha (dd/mm/yyyy): ")
f = datetime.datetime.strptime(fecha, "%d/%m/%Y")
resultado = f - datetime.timedelta(days=7)
print("Fecha 7 días antes:", resultado)
```

Explicación:

- Similar al ejercicio anterior, pero se resta un `timedelta`.

Comandos usados:

- `- datetime.timedelta(days=7)`: resta 7 días
- `.strptime()`: convierte cadena a fecha

Ejercicio 10

Enunciado:
Calcula cuántos segundos han pasado desde la medianoche de hoy.

Solución:

```
import datetime


ahora = datetime.datetime.now()
medianoche = datetime.datetime.combine(datetime.date.today(),
datetime.time(0,0,0))
segundos = (ahora - medianoche).total_seconds()
print("Segundos desde medianoche:", segundos)
```

Explicación:

- `datetime.combine()` une fecha y hora para crear un objeto completo.
- `total_seconds()` convierte un `timedelta` a segundos.

Comandos usados:

- `datetime.date.today()`: fecha de hoy
- `datetime.time(0,0,0)`: hora 00:00:00
- `(ahora - medianoche).total_seconds()`: calcula segundos transcurridos

Ejercicio 11

Enunciado: Convierte una fecha ingresada por el usuario a formato `yyyy-mm-dd`.

Solución:

```
import datetime

fecha = input("Ingresa una fecha (dd/mm/yyyy): ")
f = datetime.datetime.strptime(fecha, "%d/%m/%Y")
print("Fecha en formato ISO:", f.strftime("%Y-%m-%d"))
```

Explicación: Convierte la fecha de formato español a ISO (`yyyy-mm-dd`).

Comandos:

- `strptime()`: convierte cadena a fecha.
- `strftime("%Y-%m-%d")`: formatea fecha a año-mes-día.

Ejercicio 12

Enunciado: Imprime la fecha actual con nombre de mes y día de la semana.

Solución:

```
import datetime

hoy = datetime.datetime.now()
print(hoy.strftime("%A, %d de %B de %Y"))
```

Explicación: Usa códigos para mostrar día de la semana y mes en palabras.

Comandos:

- %A: nombre del día de la semana.
- %B: nombre del mes.
- %d y %Y: día y año.

Ejercicio 13

Enunciado: Calcula los días restantes hasta fin de año.

Solución:

```
import datetime


hoy = datetime.date.today()
fin_año = datetime.date(hoy.year, 12, 31)
faltan = (fin_año - hoy).days
print("Días hasta fin de año:", faltan)
```

Explicación: Resta la fecha actual de la última del año.

Comandos:

- `datetime.date(año, mes, día)`: crea fecha específica.
- `.days`: días de diferencia.

Ejercicio 14

Enunciado: Calcula cuántos días viviste según tu fecha de nacimiento.

Solución:

```
import datetime


nacimiento = input("Ingresa tu fecha de nacimiento (dd/mm/yyyy): ")
f_nac = datetime.datetime.strptime(nacimiento, "%d/%m/%Y")
hoy = datetime.datetime.now()
dias_vividos = (hoy - f_nac).days
print("Has vivido", dias_vividos, "días")
```

Explicación: Resta la fecha de nacimiento de la fecha actual y obtiene días.

Comandos:

- `(hoy - f_nac).days`: diferencia en días entre dos fechas.

Ejercicio 15

Enunciado: Muestra la fecha y hora actuales en formato de 12 horas con AM/PM.

Solución:

```
import datetime

ahora = datetime.datetime.now()
print("Hora actual:", ahora.strftime("%I:%M:%S %p"))
```

Explicación: Formatea la hora en 12 horas y agrega AM/PM.

Comandos:

- `%I`: hora en formato 12h.
- `%p`: AM/PM.

Ejercicio 16

Enunciado: Calcula cuántos días faltan para tu próximo cumpleaños.

Solución:

```
import datetime

cumple = input("Ingresa tu cumpleaños (dd/mm): ")
dia, mes = map(int, cumple.split("/"))
hoy = datetime.date.today()
proximo = datetime.date(hoy.year, mes, dia)
if proximo < hoy:
    proximo = datetime.date(hoy.year + 1, mes, dia)
faltan = (proximo - hoy).days
print("Días para tu próximo cumpleaños:", faltan)
```

Explicación: Calcula la fecha del próximo cumpleaños y resta la fecha actual.

Comandos:

- `split("/")`: separa día y mes.
- `map(int, ...)`: convierte cadenas a enteros.
- `if proximo < hoy`: ajusta al siguiente año si ya pasó.

Ejercicio 17

Enunciado: Calcula cuántos segundos faltan para la medianoche.

Solución:

```
import datetime


ahora = datetime.datetime.now()

medianoche = datetime.datetime.combine(datetime.date.today() +
datetime.timedelta(days=1), datetime.time(0,0,0))

faltan_segundos = (medianoche - ahora).total_seconds()

print("Segundos hasta medianoche:", int(faltan_segundos))
```

Explicación: Combina fecha de mañana con hora 0:00 y calcula la diferencia.

Comandos:

- `datetime.timedelta(days=1)`: suma un día.
- `.total_seconds()`: convierte timedelta a segundos.

Ejercicio 18

Enunciado: Muestra la fecha y hora actuales con microsegundos.

Solución:

```
import datetime

ahora = datetime.datetime.now()
print("Fecha y hora con microsegundos:", ahora.strftime("%Y-%m-%d %H:%M:%S.%f"))
```

Explicación: Incluye microsegundos en la representación de hora.

Comandos:

- `%f`: microsegundos (6 dígitos).

Ejercicio 19

Enunciado: Imprime la fecha actual y el número de semana del año.

Solución:

```
import datetime


hoy = datetime.date.today()
print("Semana del año:", hoy.isocalendar()[1])
```

Explicación: `isocalendar()` devuelve tupla `(año, semana, día)`.

Comandos:

- `isocalendar()[1]`: extrae el número de semana.

Ejercicio 20

Enunciado: Calcula la diferencia en años, meses y días entre dos fechas.

Solución:

```
import datetime
from dateutil.relativedelta import relativedelta


f1 = datetime.datetime.strptime(input("Fecha inicial (dd/mm/yyyy): "), "%d/%m/%Y")
f2 = datetime.datetime.strptime(input("Fecha final (dd/mm/yyyy): "), "%d/%m/%Y")


diff = relativedelta(f2, f1)
print(f"Diferencia: {diff.years} años, {diff.months} meses, {diff.days} días")
```

Explicación: `relativedelta` calcula diferencias exactas en años, meses y días.

Comandos:

- `from dateutil.relativedelta import relativedelta`: permite cálculo preciso entre fechas.
- `relativedelta(f2, f1)`: diferencia en años, meses y días.

Ejercicio 21

Enunciado: Calcula cuántos días hábiles hay entre dos fechas (excluyendo fines de semana).

Solución:

```
import datetime


fecha_inicio = datetime.datetime.strptime(input("Fecha inicio (dd/mm/yyyy): "), "%d/%m/%Y")
fecha_fin = datetime.datetime.strptime(input("Fecha fin (dd/mm/yyyy): "), "%d/%m/%Y")


dias_habiles = 0
fecha_actual = fecha_inicio
while fecha_actual <= fecha_fin:
    if fecha_actual.weekday() < 5:  # 0=lunes, 6=domingo
        dias_habiles += 1
    fecha_actual += datetime.timedelta(days=1)


print("Días hábiles:", dias_habiles)
```

Explicación: Recorre cada día y cuenta solo los días de lunes a viernes.

Comandos:

- `weekday() < 5`: comprueba que no sea sábado ni domingo.
- `datetime.timedelta(days=1)`: suma un día al bucle.

Ejercicio 22

Enunciado: Convierte segundos a horas, minutos y segundos.

Solución:

```
segundos_totales = int(input("Ingresa los segundos: "))

horas = segundos_totales // 3600
minutos = (segundos_totales % 3600) // 60
segundos = segundos_totales % 60

print(f"{horas} horas, {minutos} minutos, {segundos} segundos")
```

Explicación: Convierte un total de segundos a horas:minutos:segundos.

Comandos:

- `//`: división entera.
- %: módulo, obtiene residuo de la división.

Ejercicio 23

Enunciado: Imprime el calendario del mes actual.

Solución:

```
import calendar
import datetime

hoy = datetime.date.today()
print(calendar.month(hoy.year, hoy.month))
```

Explicación: Usa módulo `calendar` para imprimir el calendario del mes actual.

Comandos:

- `calendar.month(año, mes)`: imprime el mes en formato de calendario.

Ejercicio 24

Enunciado: Calcula la fecha y hora exacta dentro de 1000000 segundos desde ahora.

Solución:

```
import datetime


ahora = datetime.datetime.now()
futuro = ahora + datetime.timedelta(seconds=1000000)
print("Fecha y hora dentro de 1,000,000 segundos:", futuro)
```

Explicación: Suma un intervalo de tiempo en segundos a la fecha actual.

Comandos:

- `timedelta(seconds=n)`: representa un intervalo de n segundos.

Ejercicio 25

Enunciado: Muestra la fecha actual en formato "Miércoles, 15 de Enero de 2025, 14:35:20".

Solución:

```
import datetime

ahora = datetime.datetime.now()
print(ahora.strftime("%A, %d de %B de %Y, %H:%M:%S"))
```

Explicación: Formatea fecha y hora completas con día de la semana y nombre del mes.

Comandos:

- `%A`: día de la semana
- `%B`: mes completo
- `%H:%M:%S`: hora en formato 24h

Ejercicio 26

Enunciado: Calcula el número de días entre dos fechas ignorando los años bisiestos.

Solución:

```
import datetime

f1 = datetime.datetime.strptime(input("Fecha 1 (dd/mm/yyyy): "), "%d/%m/%Y")
f2 = datetime.datetime.strptime(input("Fecha 2 (dd/mm/yyyy): "), "%d/%m/%Y")

dias = (f2 - f1).days
print("Días entre fechas (ignorando bisiestos):", dias)
```

Explicación: Simple resta de fechas, ignorando la consideración de bisiestos ya que Python lo maneja automáticamente en `datetime`.

Comandos:

- `(f2 - f1).days`: diferencia de días entre dos fechas.

Ejercicio 27

Enunciado: Calcula cuántos viernes hay entre dos fechas.

Solución:

```
import datetime


fecha_inicio = datetime.datetime.strptime(input("Fecha inicio (dd/mm/yyyy): "), "%d/%m/%Y")
fecha_fin = datetime.datetime.strptime(input("Fecha fin (dd/mm/yyyy): "), "%d/%m/%Y")


viernes = 0
fecha_actual = fecha_inicio
while fecha_actual <= fecha_fin:
    if fecha_actual.weekday() == 4:  # Viernes
        viernes += 1
    fecha_actual += datetime.timedelta(days=1)


print("Cantidad de viernes:", viernes)
```

Explicación: Cuenta los días que corresponden al viernes entre dos fechas.

Comandos:

- `weekday() == 4`: identifica viernes.

Ejercicio 28

Enunciado: Calcula la edad exacta en años, meses y días a partir de la fecha de nacimiento.

Solución:

```
import datetime

from dateutil.relativedelta import relativedelta


nacimiento = datetime.datetime.strptime(input("Fecha de nacimiento (dd/mm/yyyy): "), "%d/%m/%Y")

hoy = datetime.datetime.now()

edad = relativedelta(hoy, nacimiento)

print(f"Tienes {edad.years} años, {edad.months} meses y {edad.days} días")
```

Explicación: `relativedelta` permite obtener diferencias exactas de años, meses y días.

Comandos:

- `relativedelta(f2, f1)`: diferencia precisa entre fechas.

Ejercicio 29

Enunciado: Imprime la hora actual en distintos husos horarios (UTC y local).

Solución:

```
from datetime import datetime, timezone

hora_utc = datetime.now(timezone.utc)
hora_local = datetime.now()

print("Hora UTC:", hora_utc)
print("Hora local:", hora_local)
```

Explicación: Usa `timezone.utc` para obtener hora UTC, y `datetime.now()` para hora local.

Comandos:

- `timezone.utc`: zona horaria UTC
- `datetime.now(timezone)`: hora según zona horaria

Ejercicio 30

Enunciado: Convierte un timestamp (segundos desde 1970) a fecha legible.

Solución:

```
import datetime


timestamp = int(input("Ingresa un timestamp: "))
fecha = datetime.datetime.fromtimestamp(timestamp)
print("Fecha correspondiente:", fecha)
```

Explicación: `fromtimestamp()` convierte segundos desde epoch a objeto datetime.

Comandos:

- `fromtimestamp(n)`: transforma segundos desde 1970 a fecha.

Ejercicio 31

Enunciado: Calcula cuántos días faltan para el próximo viernes 13.

Solución:

```
import datetime


hoy = datetime.date.today()
año = hoy.year
while True:
    for mes in range(1, 13):
        fecha = datetime.date(año, mes, 13)
        if fecha.weekday() == 4 and fecha > hoy:
            print("El próximo viernes 13 es:", fecha)
            break
    else:
        año += 1
        continue
    break
```

Explicación: Busca mes por mes hasta encontrar el próximo viernes 13 posterior a hoy.

Comandos:

- `weekday() == 4`: identifica viernes.
- `datetime.date(año, mes, día)`: crea una fecha.

Ejercicio 32

Enunciado: Calcula la duración de un evento ingresando fecha y hora de inicio y fin.

Solución:

```
import datetime


inicio = datetime.datetime.strptime(input("Inicio (dd/mm/yyyy HH:MM): "), "%d/%m/%Y %H:%M")

fin = datetime.datetime.strptime(input("Fin (dd/mm/yyyy HH:MM): "), "%d/%m/%Y %H:%M")


duracion = fin - inicio

print("Duración del evento:", duracion)
```

Explicación: Resta dos objetos datetime para obtener un timedelta.

Comandos:

- `datetime.strptime(cadena, formato)`: convierte cadena a fecha y hora.
- `fin - inicio`: timedelta entre dos fechas.

Ejercicio 33

Enunciado: Calcula el número de semanas completas entre dos fechas.

Solución:

```
import datetime

f1 = datetime.datetime.strptime(input("Fecha 1 (dd/mm/yyyy): "), "%d/%m/%Y")
f2 = datetime.datetime.strptime(input("Fecha 2 (dd/mm/yyyy): "), "%d/%m/%Y")

semanas = (f2 - f1).days // 7
print("Semanas completas entre fechas:", semanas)
```

Explicación: Divide la diferencia en días entre 7 para obtener semanas completas.

Comandos:

- `(f2 - f1).days // 7`: semanas completas.

Ejercicio 34

Enunciado: Convierte fecha y hora locales a UTC.

Solución:

```
from datetime import datetime, timezone


local = datetime.strptime(input("Fecha y hora local (dd/mm/yyyy HH:MM): "), "%d/%m/%Y %H:%M")

local_utc = local.astimezone(timezone.utc)

print("Hora en UTC:", local_utc)
```

Explicación: Convierte hora local a zona horaria UTC.

Comandos:

- `astimezone(timezone.utc)`: convierte a UTC.

Ejercicio 35

Enunciado: Calcula el tiempo transcurrido desde el inicio del año hasta hoy.

Solución:

```
import datetime

hoy = datetime.date.today()
inicio_año = datetime.date(hoy.year, 1, 1)
dias_transcurridos = (hoy - inicio_año).days
print("Días transcurridos desde el inicio del año:", dias_transcurridos)
```

Explicación: Resta la fecha de hoy con el 1 de enero del mismo año.

Comandos:

- `datetime.date(año, mes, día)`: crea una fecha específica.

Ejercicio 36

Enunciado: Calcula la cantidad de horas trabajadas en una semana ingresando horas de entrada y salida por día.

Solución:

```
import datetime


total_horas = 0
for i in range(5):
    entrada = input(f"Entrada día {i+1} (HH:MM): ")
    salida = input(f"Salida día {i+1} (HH:MM): ")
    f_entrada = datetime.datetime.strptime(entrada, "%H:%M")
    f_salida = datetime.datetime.strptime(salida, "%H:%M")
    duracion = f_salida - f_entrada
    total_horas += duracion.seconds / 3600

print("Horas trabajadas en la semana:", total_horas)
```

Explicación: Calcula horas diarias y suma total semanal.

Comandos:

- `duracion.seconds / 3600`: convierte segundos a horas.

Ejercicio 37

Enunciado: Imprime la fecha y hora actual en formato ISO 8601.

Solución:

```
import datetime


ahora = datetime.datetime.now()
print("Fecha y hora ISO 8601:", ahora.isoformat())
```

Explicación: `isoformat()` devuelve fecha y hora en estándar internacional ISO 8601.

Comandos:

- `isoformat()`: convierte objeto datetime a formato ISO.

Ejercicio 38

Enunciado: Calcula la fecha del próximo lunes a partir de hoy.

Solución:

```
import datetime

hoy = datetime.date.today()
dias_a_lunes = (0 - hoy.weekday() + 7) % 7
if dias_a_lunes == 0:
    dias_a_lunes = 7
proximo_lunes = hoy + datetime.timedelta(days=dias_a_lunes)
print("Próximo lunes:", proximo_lunes)
```

Explicación: Calcula el número de días hasta el próximo lunes usando módulo 7.

Comandos:

- `weekday()`: devuelve número de día (lunes=0).
- `% 7`: asegura que el resultado esté en rango 0-6.

Ejercicio 39

Enunciado: Calcula cuántos meses completos han pasado desde una fecha dada.

Solución:

```
from dateutil.relativedelta import relativedelta
import datetime


fecha = datetime.datetime.strptime(input("Ingresa fecha (dd/mm/yyyy): "), "%d/%m/%Y")
hoy = datetime.datetime.now()
diferencia = relativedelta(hoy, fecha)
meses = diferencia.years * 12 + diferencia.months
print("Meses completos transcurridos:", meses)
```

Explicación: Calcula años y meses usando `relativedelta` y convierte todo a meses.

Comandos:

- `relativedelta(f2, f1)`: diferencia precisa en años, meses y días.

Ejercicio 40

Enunciado: Muestra la fecha y hora actuales en tres husos horarios distintos.

Solución:

```
from datetime import datetime, timezone, timedelta

ahora_utc = datetime.now(timezone.utc)
hora_nueva_york = ahora_utc + timedelta(hours=-5)
hora_tokio = ahora_utc + timedelta(hours=9)
hora_londres = ahora_utc + timedelta(hours=0)

print("Hora UTC:", ahora_utc)
print("Hora Nueva York:", hora_nueva_york)
print("Hora Tokio:", hora_tokio)
print("Hora Londres:", hora_londres)
```

Explicación: Ajusta la hora UTC a distintos husos horarios usando `timedelta`.

Comandos:

- `timedelta(hours=n)`: suma o resta horas.
- `timezone.utc`: define la zona horaria UTC.

Ejercicio 41

Enunciado: Crea un programa que calcule cuántos días faltan para el próximo año bisiesto.

Solución:

```
import datetime


hoy = datetime.date.today()
año = hoy.year + 1
while True:
    if (año % 4 == 0 and año % 100 != 0) or (año % 400 == 0):
        bisiesto = datetime.date(año, 2, 29)
        dias_faltantes = (bisiesto - hoy).days
        print("Días hasta el próximo año bisiesto:", dias_faltantes)
        break
    año += 1
```

Explicación: Busca el próximo año bisiesto y calcula la diferencia en días desde hoy.

Comandos:

- `datetime.date(año, mes, día)`: crea una fecha.
- `(año % 4 == 0 and año % 100 != 0) or (año % 400 == 0)`: regla de año bisiesto.

Ejercicio 42

Enunciado: Calcula el tiempo transcurrido desde una fecha y hora específica hasta ahora en días, horas y minutos.

Solución:

```
import datetime


fecha_hora = datetime.datetime.strptime(input("Fecha y hora (dd/mm/yyyy HH:MM): "), "%d/%m/%Y %H:%M")
ahora = datetime.datetime.now()
diferencia = ahora - fecha_hora


dias = diferencia.days
horas = diferencia.seconds // 3600
minutos = (diferencia.seconds % 3600) // 60


print(f"Tiempo transcurrido: {dias} días, {horas} horas, {minutos} minutos")
```

Explicación: Convierte `timedelta` en días, horas y minutos.

Comandos:

- `diferencia.seconds // 3600`: obtiene horas completas.
- `(diferencia.seconds % 3600) // 60`: obtiene minutos.

Ejercicio 43

Enunciado: Crea un cronómetro que muestre segundos transcurridos hasta que el usuario presione Enter.

Solución:

```
import time


input("Presiona Enter para iniciar el cronómetro...")
inicio = time.time()
input("Presiona Enter para detener el cronómetro...")
fin = time.time()
print("Segundos transcurridos:", fin - inicio)
```

Explicación: Usa `time.time()` para medir segundos entre dos eventos.

Comandos:

- `time.time()`: devuelve segundos desde epoch (1970).

Ejercicio 44

Enunciado: Calcula el número de días que tiene cada mes del año actual.

Solución:

```
import calendar


año = int(input("Ingresa el año: "))
for mes in range(1, 13):
    dias = calendar.monthrange(año, mes)[1]
    print(f"Mes {mes} tiene {dias} días")
```

Explicación: `calendar.monthrange()` devuelve día de la semana del primer día y cantidad de días del mes.

Comandos:

- `calendar.monthrange(año, mes)[1]`: obtiene número de días del mes.

Ejercicio 45

Enunciado: Muestra la fecha actual en formato europeo, americano y ISO.

Solución:

```
import datetime

hoy = datetime.datetime.now()
formato_europeo = hoy.strftime("%d/%m/%Y")
formato_americano = hoy.strftime("%m/%d/%Y")
formato_iso = hoy.isoformat()

print("Formato europeo:", formato_europeo)
print("Formato americano:", formato_americano)
print("Formato ISO:", formato_iso)
```

Explicación: Usa `strftime` e `isoformat` para mostrar distintos formatos.

Comandos:

- `%d/%m/%Y`: día/mes/año
- `%m/%d/%Y`: mes/día/año
- `isoformat()`: formato estándar internacional

Ejercicio 46

Enunciado: Calcula el día de la semana para cualquier fecha ingresada.

Solución:

```
import datetime


fecha = datetime.datetime.strptime(input("Ingresa fecha (dd/mm/yyyy): "), "%d/%m/%Y")

dias = ["Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo"]

print("Día de la semana:", dias[fecha.weekday()])
```

Explicación: Convierte fecha a número de día de la semana y lo traduce a nombre.

Comandos:

- `weekday()`: devuelve 0-6 para lunes a domingo.

Ejercicio 47

Enunciado: Crea un programa que calcule cuántos segundos hay en un año bisiesto y en uno normal.

Solución:

```
segundos_normal = 365 * 24 * 3600
segundos_bisiesto = 366 * 24 * 3600
print("Segundos en un año normal:", segundos_normal)
print("Segundos en un año bisiesto:", segundos_bisiesto)
```

Explicación: Multiplica días por horas y segundos para calcular total.

Comandos:

- `*`: multiplicación.
- `3600`: segundos por hora.

Ejercicio 48

Enunciado: Calcula la diferencia entre dos zonas horarias dadas (por ejemplo, Tokio y Nueva York).

Solución:

```
from datetime import datetime, timedelta, timezone


tokio = datetime.now(timezone(timedelta(hours=9)))
nueva_york = datetime.now(timezone(timedelta(hours=-5)))
diferencia = tokio - nueva_york
print("Diferencia entre Tokio y Nueva York:", diferencia)
```

Explicación: Usa `timezone` y `timedelta` para calcular diferencia de horas.

Comandos:

- `timezone(timedelta(hours=n))`: define zona horaria.
- `datetime.now(timezone)`: hora en zona específica.

Ejercicio 49

Enunciado: Calcula la fecha de Pascua para un año dado usando el algoritmo de Meeus.

Solución:

```
year = int(input("Ingresa el año: "))
a = year % 19
b = year // 100
c = year % 100
d = b // 4
e = b % 4
f = (b + 8) // 25
g = (b - f + 1) // 3
h = (19*a + b - d - g + 15) % 30
i = c // 4
k = c % 4
l = (32 + 2*e + 2*i - h - k) % 7
m = (a + 11*h + 22*l) // 451
mes = (h + l - 7*m + 114) // 31
dia = ((h + l - 7*m + 114) % 31) + 1
print(f"Pascua en {year}: {dia}/{mes}/{year}")
```

Explicación: Algoritmo de Meeus para calcular fecha de Pascua en calendario gregoriano.

Comandos:

- `%`, `//`: módulo y división entera para cálculos.

Ejercicio 50

Enunciado: Crea un programa que muestre un recordatorio en una fecha y hora futura específica.

Solución:

```
import datetime

import time


recordatorio = datetime.datetime.strptime(input("Ingresa fecha y hora del recordatorio (dd/mm/yyyy HH:MM): "), "%d/%m/%Y %H:%M")

print("Esperando el recordatorio...")

while datetime.datetime.now() < recordatorio:

    time.sleep(10)

print("¡Es hora del recordatorio!")
```

Explicación: Espera hasta que la fecha y hora del recordatorio llegue, revisando cada 10 segundos.

Comandos:

- `time.sleep(n)`: pausa ejecución n segundos.
- `while datetime.datetime.now() < recordatorio`: bucle hasta la hora del recordatorio.

MANUAL DETALLADO DE EJERCICIOS DE FORMATO EN PYTHON

Bloque 1: Introducción a .format() y Conversiones (Ejercicios 1-10)

Ejercicio 1: Imprimir tu nombre

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide el nombre al usuario y lo muestra en una frase usando `.format()`.	
Solución	`nombre = input("Escribe tu nombre: ") print("Hola, {}!".format(nombre))`	
Explicación	La función **`input()`** se utiliza para pedir un dato al usuario. La cadena `"Hola, {}!".format(nombre)` reemplaza las llaves **`{}`** con el valor almacenado en la variable `nombre`. La función **`print()`** muestra el resultado.	**`{}`** → lugar donde se insertará el valor. **`.format(variable)`** → reemplaza `{}` por la variable.

Ejercicio 2: Mostrar edad

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide la edad del usuario y la muestra en una frase.	
Solución	`edad = int(input("Escribe tu edad: ")) print("Tienes {} años".format(edad))`	
Explicación	Se utiliza **`int()`** para asegurar que la entrada del usuario se convierta a un número entero. Luego, **`.format(edad)`** se encarga de insertar el valor de `edad` en la cadena de texto.	**`int()`** → convierte la entrada a número entero. **`{}`** → marcador de posición. **`.format()`** → sustituye los marcadores con valores.

Ejercicio 3: Nombre y edad con f-string

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Usa f-string para mostrar el nombre y edad en una frase.	
Solución	`nombre = input("Escribe tu nombre: ") edad = int(input("Escribe tu edad: ")) print(f"Hola {nombre}, tienes {edad} años")`	
Explicación	El prefijo **`f`** antes de la cadena (`f"..."`) permite incluir variables directamente dentro de las llaves `{}`. El comando **`{variable}`** inserta el valor de	**`f"..."`** → permite incluir variables directamente. **`{variable}`** → inserta el valor

	la variable dentro de la cadena.	de la variable.

Ejercicio 4: Número decimal con 2 decimales

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide un número decimal y muéstralo con 2 decimales.	
Solución	`numero = float(input("Escribe un número decimal: ")) print("Número con 2 decimales: {:.2f}".format(numero))`	
Explicación	La función **`float()`** convierte la entrada a un número decimal. El especificador **`:.2f`** dentro de las llaves asegura que el número flotante se muestre con exactamente 2 decimales.	**`float()`** → convierte a decimal. **`:.2f`** → formatea el número con 2 decimales.

Ejercicio 5: Alinear texto a la izquierda

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Imprime el nombre alineado a la izquierda en un espacio de 20 caracteres.	
Solución	`nombre = input("Escribe tu nombre: ") print("Nombre: {:<20} fin".format(nombre))`	
Explicación	El comando **`<`** se utiliza para alinear el texto a la izquierda. El número **`20`** define el ancho total de la cadena en 20 caracteres.	**`<`** → alinear a la izquierda. **`20`** → ancho de 20 caracteres.

Ejercicio 6: Alinear texto al centro

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Muestra el nombre centrado en 20 caracteres.	
Solución	`nombre = input("Escribe tu nombre: ") print("Nombre: {:^20} fin".format(nombre))`	
Explicación	El especificador **`^`** se usa para centrar el texto dentro del espacio definido. El valor **`20`** establece el ancho total de la cadena.	**`^`** → centrar texto. **`20`** → ancho total.

Ejercicio 7: Alinear texto a la derecha

Componente	Explicación Detallada	Comandos/ Controles Clave
Enunciado	Muestra el nombre alineado a la derecha en 20 caracteres.	
Solución	`nombre = input("Escribe tu nombre: ") print("Nombre: {:>20}fin".format(nombre))`	
Explicación	El carácter **`>`** se utiliza para alinear el texto a la derecha.	**`>`** → alinear a la derecha.

Ejercicio 8: Rellenar con ceros

Componente	Explicación Detallada	Comandos/Controles Clave

Enunciado	Pide un número y lo muestra con 5 dígitos, rellenando con ceros.	
Solución	`numero = int(input("Escribe un número: ")) print("Número: {:05}".format(numero))`	
Explicación	El comando **`0`** se utiliza para indicar que el espacio vacío debe rellenarse con ceros. El valor **`5`** define el ancho total (cantidad de dígitos).	**`0`** → rellena con ceros. **`5`** → ancho total.

Ejercicio 9: Mostrar porcentaje

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide un número decimal y lo muestra como porcentaje con 1 decimal.	
Solución	`numero = float(input("Escribe un número decimal: ")) print("Porcentaje: {:.1%}".format(numero))`	
Explicación	El especificador **`.1%`** formatea el número como un porcentaje, asegurando que se muestre con un solo decimal.	**`.1%`** → formatea el número como porcentaje con 1 decimal.

Ejercicio 10: Varias variables

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide nombre y edad y muéstralos en la misma frase usando `.format()`.	
Solución	`nombre = input("Escribe tu nombre: ") edad = int(input("Escribe tu edad: ")) print("Nombre: {}, Edad: {}".format(nombre, edad))`	
Explicación	La función **`.format(nombre, edad)`** reemplaza secuencialmente los marcadores **`{}`** de la cadena con el valor de cada variable en el orden especificado.	**`.format(variable1, variable2)`** → reemplaza los `{}` por cada variable en orden.

Bloque 2: Formato Avanzado, Alineación y Tipos Numéricos (Ejercicios 11-20)

Ejercicio 11: Formato con índice

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide nombre y apellido y muéstralos usando `.format()` con índices.	
Solución	`nombre = input("Nombre: ") apellido = input("Apellido: ") print("Apellido: {1}, Nombre: {0}".format(nombre, apellido))`	

Explicación	El uso de **{0}** y **{1}** dentro de la cadena indica el orden posicional de las variables pasadas a `.format()`. En este caso, se imprime primero el apellido (`{1}`) y luego el nombre (`{0}`).	**{0}**, **{1}** → indican el orden de las variables en `.format()`.

Ejercicio 12: Formato con nombre de variables

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide nombre y edad, muestra “Nombre: X, Edad: Y” usando nombres de variables en `.format()`.	
Solución	`nombre = input("Nombre: ") edad = int(input("Edad: ")) print("Nombre: {n}, Edad: {e}".format(n=nombre, e=edad))`	
Explicación	Los marcadores **{n}** y **{e}** hacen referencia a las variables pasadas como argumentos con nombre (`n=nombre`, `e=edad`) a la función `.format()`.	**{n}**, **{e}** → hacen referencia a variables pasadas como argumentos con nombre.

Ejercicio 13: Decimal con 3 cifras (f-string)

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide un número decimal y muéstralo con 3 cifras decimales usando f-string.	
Solución	`numero = float(input("Número decimal: ")) print(f"Número con 3 decimales: {numero:.3f}")`	
Explicación	Se utiliza el especificador **:.3f** para formatear el número flotante, garantizando 3 decimales. El prefijo **f"..."** permite usar la variable directamente.	**:.3f** → número flotante con 3 decimales. **f"..."** → permite usar variables directamente.

Ejercicio 14: Mostrar número en notación científica

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide un número y muéstralo en notación científica con 2 decimales.	
Solución	`numero = float(input("Número: ")) print("Notación científica: {:.2e}".format(numero))`	
Explicación	El especificador **:.2e** convierte el número a notación científica, mostrando dos decimales en el mantisa.	**:.2e** → número en notación científica con 2 decimales.

Ejercicio 15: Alinear y rellenar con espacios

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide el nombre y muéstralo alineado a la derecha en 20 caracteres, rellenando con puntos.	
Solución	`nombre = input("Nombre: ") print("Nombre: {:->20}".format(nombre))`	

Explicación	El carácter **-** se especifica como el carácter de relleno (guiones). El comando **>** alinea el texto a la derecha. **20** define el ancho total.	**-** → rellena con guiones. **>** → alinea a la derecha. **20** → ancho total.

Ejercicio 16: Tabla de múltiplos

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Muestra los números del 1 al 5 y su cuadrado en columnas alineadas.	
Solución	`for i in range(1, 6): print("{:<5} {:<5}".format(i, i**2))`	
Explicación	Se usa un bucle **for** para repetir la impresión. El especificador **<5** define un ancho de 5 caracteres con alineación a la izquierda. La expresión **i**2** calcula el cuadrado del número **i**.	**for** → bucle para repetir. **<5** → ancho de 5 caracteres, alineado a la izquierda. **i**2** → cuadrado del número.

Ejercicio 17: Rellenar con ceros y alinear

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide un número y muéstralo con 8 dígitos, rellenando con ceros y alineado a la derecha.	
Solución	`numero = int(input("Número: ")) print("{:0>8}".format(numero))`	
Explicación	El comando **0>** indica rellenar con ceros y alinear el resultado a la derecha. El número **8** establece el ancho total.	**0>** → rellena con ceros y alinea a la derecha. **8** → ancho total.

Ejercicio 18: Porcentaje con 2 decimales (f-string)

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide un número decimal y muéstralo como porcentaje con 2 decimales usando f-string.	
Solución	`numero = float(input("Número decimal: ")) print(f"Porcentaje: {numero:.2%}")`	
Explicación	El especificador **:.2%** convierte el valor a porcentaje y asegura que se muestren 2 decimales.	**:.2%** → convierte a porcentaje y muestra 2 decimales.

Ejercicio 19: Combinando variables y texto

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide nombre, edad y ciudad. Muestra: “Nombre X, Edad Y, Ciudad Z” usando `.format()`.	
Solución	`nombre = input("Nombre: ") edad = int(input("Edad: ")) ciudad = input("Ciudad: ") print("Nombre: {}, Edad: {}, Ciudad: {}".format(nombre, edad, ciudad))`	
Explicación	El método **.format()** reemplaza los marcadores **{}** en el **.format()** →	

	orden en que se proporcionan las variables (`nombre`, `edad`, `ciudad`).	reemplaza los `{}` en orden con cada variable.

Ejercicio 20: Tabla con encabezados

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Muestra una tabla de productos y precios alineados con encabezados.	
Solución	`print("{:<10} {:>5}".format("Producto", "Precio")) for p, pr in zip(productos, precios): print("{:<10} {:>5.2f}".format(p, pr))`	
Explicación	**`<10`** establece un ancho de 10 y alinea a la izquierda para los productos. **`>5.2f`** establece un ancho de 5, asegura 2 decimales (`.2f`), y alinea a la derecha (>) para los precios. **`zip()`** combina las listas para iterar sobre ellas simultáneamente.	**`<10`** → ancho 10, alineado a la izquierda. **`>5.2f`** → ancho 5, 2 decimales, alineado a la derecha. **`zip()`** → combina listas para iterar.

Bloque 3: Formatos Especiales y Bases Numéricas (Ejercicios 21-30)

Ejercicio 21: Formateo con signos

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide un número (positivo o negativo) y muéstralo siempre con el signo usando `.format()`.	
Solución	`numero = float(input("Número: ")) print("Número con signo: {:+}".format(numero))`	
Explicación	El comando **+** dentro del especificador de formato fuerza a que se muestre siempre el signo, ya sea `+` o `-`.	**`+`** → fuerza mostrar el signo (+ o -).

Ejercicio 22: Número con espacio para signo

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Muestra un número dejando un espacio si es positivo, y con `-` si es negativo.	
Solución	`numero = float(input("Número: ")) print("Número: {: }".format(numero))`	
Explicación	El especificador **:** (espacio) deja un espacio en blanco reservado para el signo cuando el número es positivo.	**`:`** → deja un espacio para números positivos.

Ejercicio 23: Mostrar número en hexadecimal

Componente	Explicación Detallada	Comandos/Controles Clave

Enunciado	Pide un número entero y muéstralo en hexadecimal usando `.format()`.	
Solución	`numero = int(input("Número entero: ")) print("Hexadecimal: {:x}".format(numero))`	
Explicación	El comando **`x`** formatea el número entero en su representación hexadecimal, utilizando letras minúsculas.	**`x`** → formatea el número en hexadecimal (minúsculas).

Ejercicio 24: Mostrar número en octal

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide un número entero y muéstralo en octal usando `.format()`.	
Solución	`numero = int(input("Número entero: ")) print("Octal: {:o}".format(numero))`	
Explicación	El comando **`o`** formatea el número entero en su representación octal.	**`o`** → formatea el número en octal.

Ejercicio 25: Número en binario

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide un número entero y muéstralo en binario usando f-string.	
Solución	`numero = int(input("Número entero: ")) print(f"Binario: {numero:b}")`	
Explicación	El especificador **`:b`** formatea el número en su representación binaria.	**`:b`** → formatea el número en binario.

Ejercicio 26: Formato con relleno y ancho

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide un número y muéstralo centrado en 10 caracteres, rellenando con `*`.	
Solución	`numero = int(input("Número: ")) print("{:*^10}".format(numero))`	
Explicación	El carácter **`*`** es el carácter de relleno. El comando **`^10`** indica que el contenido debe estar centrado (^) en un ancho de 10 caracteres.	**`*`** → carácter de relleno. **`^10`** → ancho 10, centrado.

Ejercicio 27: Tabla con alineación y decimales

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Crea una tabla de 3 productos con precios, mostrando los precios con 2 decimales alineados a la derecha.	
Solución	`print("{:<10} {:>7}".format("Producto", "Precio")) for p, pr in zip(productos, precios):`	

	`print("{:<10} {:>7.2f}".format(p, pr))`	
Explicación	**`<10`** alinea el texto (producto) a la izquierda. **`>7.2f`** alinea el número (precio) a la derecha, fija el ancho a 7 y asegura 2 decimales.	**`<10`** → texto alineado a la izquierda. **`>7.2f`** → número alineado a la derecha, 2 decimales.

Ejercicio 28: F-string con expresiones

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide un número y muestra su cuadrado y cubo usando f-string con expresiones.	
Solución	`numero = int(input("Número: "))` `print(f"Cuadrado: {numero**2}, Cubo: {numero**3}")`	
Explicación	Las expresiones **`{numero**2}`** y **`{numero**3}`** son evaluadas directamente dentro del f-string para calcular el cuadrado y el cubo, respectivamente.	**`{expresión}`** → evalúa la expresión dentro del f-string.

Ejercicio 29: Alineación de lista

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Imprime los números del 1 al 5 en una columna de ancho 3, alineados a la derecha.	
Solución	`for i in range(1, 6):` `print("{:>3}".format(i))`	
Explicación	El especificador **`>3`** define un ancho de 3 caracteres y alinea el contenido a la derecha.	**`>3`** → ancho 3, alineado a la derecha.

Ejercicio 30: Tabla con encabezados y formato mixto

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Muestra una tabla con productos, cantidad y precio unitario, formateando cantidades enteras y precios con 2 decimales.	
Solución	`print("{:<10} {:>5} {:>7}".format("Producto", "Cant", "Precio")) for p, c, pr in zip(productos, cantidades, precios): print("{:<10} {:>5} {:>7.2f}".format(p, c, pr))`	
Explicación	**`<10`** para el texto alineado a la izquierda. **`>5`** para los enteros alineados a la derecha. **`>7.2f`** para los flotantes alineados a la derecha con 2 decimales.	**`<10`** → texto alineado a la izquierda. **`>5`** → enteros alineados a la derecha. **`>7.2f`** → flotantes alineados a la derecha con 2 decimales.

Bloque 4: Tablas Complejas y Relleno Personalizado (Ejercicios 31-40)

Ejercicio 31: Tabla con alineación mixta y encabezados

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Crea una tabla con nombres, calificaciones y promedio, alineando texto a la izquierda y números a la derecha con 2 decimales.	
Solución	`print("{:<10} {:>10}".format("Estudiante", "Calificación")) for e, c in zip(estudiantes, calificaciones): print("{:<10} {:>10.2f}".format(e, c))`	
Explicación	**`<10`** alinea el texto a la izquierda. **`>10.2f`** alinea el número a la derecha, en ancho 10, y muestra 2 decimales.	**`<10`** → texto alineado a la izquierda. **`>10.2f`** → número alineado a la derecha, con 2 decimales.

Ejercicio 32: Tabla con relleno de puntos

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Imprime nombres de productos y precios con puntos entre ellos para alineación visual.	
Solución	`for p, pr in zip(productos, precios): print("{:.<15} {:.2f}".format(p, pr))`	
Explicación	El comando **`.<15`** utiliza el punto (`.`) como carácter de relleno y alinea a la izquierda (<), completando hasta un ancho de 15 caracteres. **`:.2f`** asegura 2 decimales para el precio.	**`.<15`** → rellena con puntos hasta completar 15 caracteres.

Ejercicio 33: Números con relleno y signo

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Muestra números positivos y negativos siempre con signo y relleno de espacios a la derecha.	
Solución	`numeros = [5, -3, 12] for n in numeros: print("{:>+5}".format(n))`	
Explicación	El especificador **`+`** obliga a mostrar siempre el signo (para positivos y negativos). **`>5`** establece un ancho total de 5, alineado a la derecha.	**`+`** → muestra siempre el signo. **`>5`** → ancho total 5, alineado a la derecha.

Ejercicio 34: Reporte de ventas con porcentaje

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Muestra ventas y porcentaje del total, formateando con 1 decimal.	

Solución	`for p, v in zip(productos, ventas): porcentaje = v / total print(f"{p}: {v} ventas ({porcentaje:.1%})")`	
Explicación	El comando **`:.1%`** convierte la proporción a porcentaje con un decimal. Se utiliza un f-string, lo que permite incluir variables y expresiones directamente dentro de la cadena.	**`:.1%`** → convierte a porcentaje con 1 decimal. **`f-string`** → permite incluir variables y expresiones directamente.

Ejercicio 35: Formato con relleno y centrado

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide un texto y muéstralo centrado en 30 caracteres, rellenando con `-`.	
Solución	`texto = input("Texto: ") print("{:-^30}".format(texto))`	
Explicación	El especificador **`-^30`** indica que el carácter de relleno es el guion (`-`), el texto debe estar centrado (^), y el ancho total es 30.	**`-^30`** → rellena con `-` y centra el texto en ancho 30.

Ejercicio 36: Mostrar números alineados en columnas

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Imprime los números del 1 al 10 y su doble en columnas alineadas.	
Solución	`for i in range(1, 11): print("{:<3} {:>3}".format(i, i*2))`	
Explicación	**`<3`** define un ancho de 3, alineado a la izquierda. **`>3`** define un ancho de 3, alineado a la derecha.	**`<3`** → ancho 3, alineado a la izquierda. **`>3`** → ancho 3, alineado a la derecha.

Ejercicio 37: Combinación de texto y número con relleno

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Muestra un nombre y un número con relleno de `.` para alinear a la derecha.	
Solución	`nombre = input("Nombre: ") numero = int(input("Número: ")) print("{:.<20} {:>5}".format(nombre, numero))`	
Explicación	**`.<20`** rellena con puntos (`.`) hasta alcanzar un ancho de 20 caracteres. **`>5`** alinea el número a la derecha en un ancho de 5.	**`.<20`** → rellena con puntos hasta ancho 20. **`>5`** → número alineado a la derecha.

Ejercicio 38: Tabla con f-strings y expresiones

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Crea una tabla con números y sus cuadrados y cubos usando f-strings.	
Solución	`for i in range(1, 6):`	

	`print(f"{i:<3} {i**2:<4} {i**3:<5}")`	
Explicación	Se utilizan expresiones como **`{i**2}`** y **`{i**3}`** directamente dentro del f-string. **`<3`**, **`<4`**, **`<5`** establecen el ancho y la alineación a la izquierda para cada columna.	**`<N`** → ancho y alineación a la izquierda. **`{expresión}`** → expresiones dentro de f-string.

Ejercicio 39: Mostrar número en hexadecimal y binario

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide un número entero y muéstralo en decimal, hexadecimal y binario alineados.	
Solución	`print("{:<5} {:<5} {:<8}".format(numero, hex(numero)[2:], bin(numero)[2:]))`	
Explicación	Las funciones **`hex()`** y **`bin()`** convierten el número a sus bases respectivas. El uso de **`[2:]`** se emplea para eliminar los prefijos por defecto (`0x` para hexadecimal y `0b` para binario). **`<5`** y **`<8`** definen la alineación en columnas.	**`hex(numero)[2:]`** → hexadecimal sin prefijo `0x`. **`bin(numero)[2:]`** → binario sin prefijo `0b`. **`<5`**, **`<8`** → alineación en columnas.

Ejercicio 40: Tabla con encabezados y relleno

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Crea una tabla de estudiantes, notas y porcentaje usando relleno de espacios y decimales.	
Solución	`print("{:<10} {:>6} {:>8}".format("Estudiante", "Nota", "% del total")) for e, n in zip(estudiantes, notas): porcentaje = n / total print("{:<10} {:>6.1f} {:>8.2%}".format(e, n, porcentaje))`	
Explicación	**`<10`** alinea el nombre a la izquierda. **`>6.1f`** alinea la nota a la derecha con 1 decimal. **`>8.2%`** alinea el porcentaje a la derecha, con un ancho de 8, y muestra 2 decimales.	**`>6.1f`** → nota con 1 decimal alineada a la derecha. **`>8.2%`** → porcentaje con 2 decimales, alineado a la derecha.

Bloque 5: Formato de Moneda y Reportes (Ejercicios 41-50)

Ejercicio 41: Formato de moneda

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide un precio y muéstralo como moneda con 2 decimales y el símbolo $..	
Solución	`precio = float(input("Precio: ")) print("Precio: $ {:.2f}".format(precio))`	

Explicación	**`:.2f`** asegura que el número se muestre con 2 decimales. El símbolo **`$`** se incluye directamente en la cadena fuera del marcador de formato.	**`:.2f`** → 2 decimales. **`$`** → se muestra directamente como símbolo de moneda.

Ejercicio 42: Tabla de productos con moneda

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Crea una tabla de productos y precios en formato de moneda alineados.	
Solución	`print("{:<10} {:>10}".format("Producto", "Precio")) for p, pr in zip(productos, precios): print("{:<10} ${:>8.2f}".format(p, pr))`	
Explicación	**`<10`** alinea el producto a la izquierda. **`$>8.2f`** inserta el símbolo de dólar y luego alinea el precio a la derecha (`>`) con un ancho de 8 y 2 decimales (`.2f`).	**`<10`** → producto alineado a la izquierda. **`>8.2f`** → precio alineado a la derecha con 2 decimales.

Ejercicio 43: Tabla con relleno de guiones

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Muestra nombres y precios con guiones como separadores hasta 20 caracteres.	
Solución	`for p, pr in zip(productos, precios): print("{:-<15} ${:>5.2f}".format(p, pr))`	
Explicación	**`-<15`** rellena con guiones (`-`) y alinea a la izquierda (`<`) hasta un ancho de 15 caracteres. **`>5.2f`** formatea el precio, alineándolo a la derecha con 2 decimales.	**`:-<15`** → rellena con guiones hasta 15 caracteres. **`>5.2f`** → precio alineado a la derecha con 2 decimales.

Ejercicio 44: Tabla con f-string y porcentaje

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Muestra ventas de productos y porcentaje del total usando f-strings.	
Solución	`for p, v in zip(productos, ventas): print(f"{p:<5} {v:>5} ({v/total:.1%})")`	
Explicación	Se usa **`<5`** para el producto (alineado a la izquierda). Se usa **`>5`** para las ventas (alineadas a la derecha). **`:.1%`** se aplica dentro del f-string para formatear el porcentaje con 1 decimal.	**`<5`** → producto alineado a la izquierda. **`>5`** → ventas alineadas a la derecha. **`:.1%`** → porcentaje con 1 decimal.

Ejercicio 45: Tabla de calificaciones con promedio

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Muestra estudiantes, 3 notas y el promedio con decimales alineados.	

Solución	`print("{:<10} {:>5} {:>5} {:>5} {:>7}".format("Estudiante", "N1", "N2", "N3", "Prom")) for e, n in zip(estudiantes, notas): promedio = sum(n)/3 print("{:<10} {:>5} {:>5} {:>5} {:>7.2f}".format(e, n, n, n, promedio))`	
Explicación	**`<10`** alinea el nombre del estudiante a la izquierda. **`>5`** se usa para alinear las notas individuales a la derecha. **`>7.2f`** alinea el promedio a la derecha, con ancho 7 y 2 decimales.	**`<10`** → estudiante alineado a la izquierda. **`>5`** → notas alineadas a la derecha. **`>7.2f`** → promedio con 2 decimales.

Ejercicio 46: Tabla de inventario con relleno

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Imprime productos, cantidades y precios con relleno de puntos para alinear.	
Solución	`for p, c, pr in zip(productos, cantidades, precios): print("{:.<10} {:>3} ${:>6.2f}".format(p, c, pr))`	
Explicación	**`.<10`** rellena con puntos (`.`) hasta un ancho de 10 caracteres. **`>3`** alinea la cantidad a la derecha. **`>6.2f`** alinea el precio a la derecha, con 2 decimales.	**`.<10`** → rellena con puntos hasta 10 caracteres. **`>3`** → cantidad alineada a la derecha. **`>6.2f`** → precio con 2 decimales alineado a la derecha.

Ejercicio 47: Formato mixto con f-string

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Pide nombre, edad y altura y muéstralos en una frase usando f-string con 1 decimal en altura.	
Solución	`nombre = input("Nombre: ") edad = int(input("Edad: ")) altura = float(input("Altura (m): ")) print(f"{nombre} tiene {edad} años y mide {altura:.1f} m")`	
Explicación	El f-string permite la mezcla fácil de texto y variables. **`:.1f`** se utiliza para formatear la variable `altura` con un solo decimal.	**`:.1f`** → altura con 1 decimal. **`f-string`** → permite mezclar texto y variables fácilmente.

Ejercicio 48: Alinear y centrar títulos

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Muestra un título centrado con ancho 40 y relleno de `*`.	
Solución	`titulo = input("Título: ") print("{:*^40}".format(titulo))`	
Explicación	El especificador **`*^40`** indica que el carácter de	**`*^40`** → centra el texto en 40

	relleno es `*`, el texto debe estar centrado (`^`), y el ancho total es 40.	caracteres con `*` de relleno.

Ejercicio 49: Tabla con números y porcentajes

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Muestra productos, ventas y porcentaje del total en columnas alineadas.	
Solución	`print("{:<10} {:>6} {:>8}".format("Producto", "Ventas", "% Total")) for p, v in zip(productos, ventas): print("{:<10} {:>6} {:>7.2%}".format(p, v, v/total))`	
Explicación	**`<10`** alinea el producto a la izquierda. **`>6`** alinea las ventas a la derecha. **`>7.2%`** alinea el porcentaje a la derecha, con un ancho de 7 y 2 decimales.	**`<10`** → producto alineado a la izquierda. **`>6`** → ventas alineadas a la derecha. **`>7.2%`** → porcentaje con 2 decimales.

Ejercicio 50: Tabla completa de inventario

Componente	Explicación Detallada	Comandos/Controles Clave
Enunciado	Imprime productos, cantidad, precio unitario y total (cantidad × precio) alineados y formateados como moneda.	
Solución	`print("{:<10} {:>8} {:>10} {:>10}".format("Producto", "Cantidad", "Precio", "Total")) for p, c, pr in zip(productos, cantidades, precios): total = c * pr print("{:<10} {:>8} ${:>9.2f} ${:>9.2f}".format(p, c, pr, total))`	
Explicación	**`c * pr`** calcula el total por producto mediante multiplicación. **`<10`** alinea el producto a la izquierda. **`>8`** alinea la cantidad entera a la derecha. **`>9.2f`** aplica 2 decimales y alinea tanto el precio unitario como el total a la derecha, con un ancho de 9.	**`c * pr`** → calcula el total por producto. **`>9.2f`** → precio y total con 2 decimales, alineados a la derecha. **`{:<10}`** → producto alineado a la izquierda.

Ejercicio 1: Imprimir tu nombre

Enunciado:
Pide al usuario su nombre y muéstralo en una frase usando `.format()`.

Solución:

```
nombre = input("Escribe tu nombre: ")
print("Hola, {}!".format(nombre))
```

Explicación:

- `input()` pide un dato al usuario.
- `"Hola, {}!".format(nombre)` reemplaza `{}` con `nombre`.
- `print()` muestra el resultado.

Comandos de FORMAT:

- `{}` → lugar donde se insertará el valor
- `.format(variable)` → reemplaza `{}` por la variable

Ejercicio 2: Mostrar edad

Enunciado:
Pide la edad del usuario y muéstrala en una frase.

Solución:

```
edad = int(input("Escribe tu edad: "))
print("Tienes {} años".format(edad))
```

Explicación:

- `int()` convierte la entrada a número entero
- `.format(edad)` inserta el valor de `edad` en la cadena

Comandos de FORMAT:

- `{}` → marcador de posición
- `.format()` → sustituye los marcadores con valores

Ejercicio 3: Nombre y edad con f-string

Enunciado:
Usa f-string para mostrar tu nombre y edad en una frase.

Solución:

```
nombre = input("Escribe tu nombre: ")
edad = int(input("Escribe tu edad: "))
print(f"Hola {nombre}, tienes {edad} años")
```

Explicación:

- `f"..."` permite incluir variables directamente
- `{variable}` inserta el valor de la variable

Ejercicio 4: Número decimal con 2 decimales

Enunciado:
Pide un número decimal y muéstralo con 2 decimales.

Solución:

```
numero = float(input("Escribe un número decimal: "))
print("Número con 2 decimales: {:.2f}".format(numero))
```

Explicación:

- `float()` convierte a decimal
- `:.2f` → número con 2 decimales

Ejercicio 5: Alinear texto a la izquierda

Enunciado:
Imprime tu nombre alineado a la izquierda en 20 caracteres.

Solución:

```
nombre = input("Escribe tu nombre: ")
print("Nombre: {:<20}fin".format(nombre))
```

Explicación:

- `<` → alinear a la izquierda
- `20` → ancho de 20 caracteres

Ejercicio 6: Alinear texto al centro

Enunciado:

Muestra tu nombre centrado en 20 caracteres.

Solución:

```
nombre = input("Escribe tu nombre: ")
print("Nombre: {:^20}fin".format(nombre))
```

Explicación:

- `^` → centrar texto
- `20` → ancho total

Ejercicio 7: Alinear texto a la derecha

Enunciado:

Muestra tu nombre alineado a la derecha en 20 caracteres.

Solución:

```
nombre = input("Escribe tu nombre: ")
print("Nombre: {:>20}fin".format(nombre))
```

Explicación:

- `>` → alinear a la derecha

Ejercicio 8: Rellenar con ceros

Enunciado:
Pide un número y muéstralo con 5 dígitos, rellenando con ceros.

Solución:

```
numero = int(input("Escribe un número: "))
print("Número: {:05}".format(numero))
```

Explicación:

- `0` → rellena con ceros
- `5` → ancho total

Ejercicio 9: Mostrar porcentaje

Enunciado:
Pide un número decimal y muéstralo como porcentaje con 1 decimal.

Solución:

```
numero = float(input("Escribe un número decimal: "))
print("Porcentaje: {:.1%}".format(numero))
```

Explicación:

- `.1%` → formatea número como porcentaje con 1 decimal

Ejercicio 10: Varias variables

Enunciado:
Pide nombre y edad y muéstralos en la misma frase usando `.format()`.

Solución:

```
nombre = input("Escribe tu nombre: ")
edad = int(input("Escribe tu edad: "))
print("Nombre: {}, Edad: {}".format(nombre, edad))
```

Explicación:

- `.format(nombre, edad)` → reemplaza los `{}` por cada variable en orden

Ejercicio 11: Formato con índice

Enunciado:
Pide nombre y apellido y muéstralos usando `.format()` con índices.

Solución:

```
nombre = input("Nombre: ")
apellido = input("Apellido: ")
print("Apellido: {1}, Nombre: {0}".format(nombre, apellido))
```

Explicación:

- `{0}` y `{1}` indican el orden de las variables en `.format()`
- Imprime primero el apellido, luego el nombre

Ejercicio 12: Formato con nombre de variables

Enunciado:
Pide nombre y edad, muestra “Nombre: X, Edad: Y” usando nombres de variables en `.format()`.

Solución:

```
nombre = input("Nombre: ")
edad = int(input("Edad: "))
print("Nombre: {n}, Edad: {e}".format(n=nombre, e=edad))
```

Explicación:

- `{n}` y `{e}` hacen referencia a las variables pasadas como argumentos con nombre

---

Ejercicio 13: Decimal con 3 cifras

Enunciado:
Pide un número decimal y muéstralo con 3 cifras decimales usando f-string.

Solución:

```
numero = float(input("Número decimal: "))
print(f"Número con 3 decimales: {numero:.3f}")
```

Explicación:

- `:.3f` → número flotante con 3 decimales
- `f"..."` permite usar variables directamente

Ejercicio 14: Mostrar número en notación científica

Enunciado:
Pide un número y muéstralo en notación científica con 2 decimales.

Solución:

```
numero = float(input("Número: "))
print("Notación científica: {:.2e}".format(numero))
```

Explicación:

- `:.2e` → número en notación científica con 2 decimales

Ejercicio 15: Alinear y rellenar con espacios

Enunciado:
Pide tu nombre y muéstralo alineado a la derecha en 20 caracteres, rellenando con puntos.

Solución:

```
nombre = input("Nombre: ")
print("Nombre: {:->20}".format(nombre))
```

Explicación:

- `-` → rellena con guiones
- `>` → alinea a la derecha
- `20` → ancho total

Ejercicio 16: Tabla de múltiplos

Enunciado:
Muestra los números del 1 al 5 y su cuadrado en columnas alineadas.

Solución:

```
for i in range(1, 6):
    print("{:<5} {:<5}".format(i, i**2))
```

Explicación:

- `<5` → ancho de 5 caracteres, alineado a la izquierda
- `i**2` → cuadrado del número
- `for` → bucle para repetir la impresión

Ejercicio 17: Rellenar con ceros y alinear

Enunciado:
Pide un número y muéstralo con 8 dígitos, rellenando con ceros y alineado a la derecha.

Solución:

```
numero = int(input("Número: "))
print("{:0>8}".format(numero))
```

Explicación:

- `0>` → rellena con ceros a la derecha
- `8` → ancho total

Ejercicio 18: Porcentaje con 2 decimales

Enunciado:
Pide un número decimal y muéstralo como porcentaje con 2 decimales usando f-string.

Solución:

```
numero = float(input("Número decimal: "))
print(f"Porcentaje: {numero:.2%}")
```

Explicación:

- `:.2%` → convierte a porcentaje y muestra 2 decimales

Ejercicio 19: Combinando variables y texto

Enunciado:
Pide nombre, edad y ciudad. Muestra: “Nombre X, Edad Y, Ciudad Z” usando `.format()`.

Solución:

```
nombre = input("Nombre: ")
edad = int(input("Edad: "))
ciudad = input("Ciudad: ")
print("Nombre: {}, Edad: {}, Ciudad: {}".format(nombre, edad, ciudad))
```

Explicación:

- `.format()` reemplaza los `{}` en orden con cada variable

Ejercicio 20: Tabla con encabezados

Enunciado:
Muestra una tabla de productos y precios alineados con encabezados.

Solución:

```
productos = ["Manzana", "Pan", "Leche"]
precios = [0.5, 1.2, 0.9]

print("{:<10} {:>5}".format("Producto", "Precio"))
for p, pr in zip(productos, precios):
    print("{:<10} {:>5.2f}".format(p, pr))
```

Explicación:

- `<10` → ancho 10, alineado a la izquierda
- `>5.2f` → ancho 5, 2 decimales, alineado a la derecha
- `zip()` combina listas para iterar al mismo tiempo

Ejercicio 21: Formateo con signos

Enunciado:
Pide un número positivo o negativo y muéstralo siempre con el signo usando `.format()`.

Solución:

```
numero = float(input("Número: "))
print("Número con signo: {:+}".format(numero))
```

Explicación:

- `+` → fuerza mostrar el signo (`+` o `-`)

Ejercicio 22: Número con espacio para signo

Enunciado:
Muestra un número con espacio si es positivo y con `-` si es negativo.

Solución:

```
numero = float(input("Número: "))
print("Número: {: }".format(numero))
```

Explicación:

- `:` → deja un espacio para números positivos

Ejercicio 23: Mostrar número en hexadecimal

Enunciado:
Pide un número entero y muéstralo en hexadecimal usando `.format()`.

Solución:

```
numero = int(input("Número entero: "))
print("Hexadecimal: {:x}".format(numero))
```

Explicación:

- `x` → formatea el número en hexadecimal (minúsculas)

Ejercicio 24: Mostrar número en octal

Enunciado:
Pide un número entero y muéstralo en octal usando `.format()`.

Solución:

```
numero = int(input("Número entero: "))
print("Octal: {:o}".format(numero))
```

Explicación:

- `o` → formatea el número en octal

Ejercicio 25: Número en binario

Enunciado:
Pide un número entero y muéstralo en binario usando f-string.

Solución:

```
numero = int(input("Número entero: "))
print(f"Binario: {numero:b}")
```

Explicación:

- `:b` → formatea el número en binario

Ejercicio 26: Formato con relleno y ancho

Enunciado:
Pide un número y muéstralo centrado en 10 caracteres, rellenando con `*`.

Solución:

```
numero = int(input("Número: "))
print("{:*^10}".format(numero))
```

Explicación:

- `*` → carácter de relleno
- `^10` → ancho 10, centrado

Ejercicio 27: Tabla con alineación y decimales

Enunciado:
Crea una tabla de 3 productos con precios, mostrando los precios con 2 decimales alineados a la derecha.

Solución:

```
productos = ["Arroz", "Huevos", "Queso"]
precios = [1.2, 2.5, 3.75]

print("{:<10} {:>7}".format("Producto", "Precio"))
for p, pr in zip(productos, precios):
    print("{:<10} {:>7.2f}".format(p, pr))
```

Explicación:

- `<10` → texto alineado a la izquierda
- `>7.2f` → número alineado a la derecha, 2 decimales

Ejercicio 28: F-string con expresiones

Enunciado:
Pide un número y muestra su cuadrado y cubo usando f-string con expresiones.

Solución:

```
numero = int(input("Número: "))
print(f"Cuadrado: {numero**2}, Cubo: {numero**3}")
```

Explicación:

- `{numero**2}` → evalúa la expresión dentro del f-string
- `{numero**3}` → lo mismo para el cubo

Ejercicio 29: Alineación de lista

Enunciado:
Imprime los números del 1 al 5 en una columna de ancho 3, alineados a la derecha.

Solución:

```
for i in range(1, 6):
    print("{:>3}".format(i))
```

Explicación:

- `>3` → ancho 3, alineado a la derecha

Ejercicio 30: Tabla con encabezados y formato mixto

Enunciado:
Muestra una tabla con productos, cantidad y precio unitario, formateando cantidades enteras y precios con 2 decimales.

Solución:

```
productos = ["Leche", "Pan", "Manzana"]
cantidades = [2, 5, 10]
precios = [1.5, 0.8, 0.3]

print("{:<10} {:>5} {:>7}".format("Producto", "Cant", "Precio"))
for p, c, pr in zip(productos, cantidades, precios):
    print("{:<10} {:>5} {:>7.2f}".format(p, c, pr))
```

Explicación:

- `<10` → texto alineado a la izquierda
- `>5` → enteros alineados a la derecha
- `>7.2f` → flotantes alineados a la derecha con 2 decimales

Ejercicio 31: Tabla con alineación mixta y encabezados

Enunciado:
Crea una tabla con nombres de estudiantes, calificaciones y promedio, alineando texto a la izquierda y números a la derecha con 2 decimales.

Solución:

```
estudiantes = ["Ana", "Luis", "Carlos"]
calificaciones = [8.5, 9.2, 7.8]

print("{:<10} {:>10}".format("Estudiante", "Calificación"))
for e, c in zip(estudiantes, calificaciones):
    print("{:<10} {:>10.2f}".format(e, c))
```

Explicación:

- `<10` → texto alineado a la izquierda
- `>10.2f` → número alineado a la derecha, con 2 decimales

Ejercicio 32: Tabla con relleno de puntos

Enunciado:
Imprime nombres de productos y precios con puntos entre ellos para alineación visual.

Solución:

```
productos = ["Leche", "Pan", "Queso"]
precios = [1.5, 0.8, 2.3]

for p, pr in zip(productos, precios):
    print("{:.<15} {:.2f}".format(p, pr))
```

Explicación:

- `. <15` → rellena con puntos hasta completar 15 caracteres

Ejercicio 33: Números con relleno y signo

Enunciado:
Muestra números positivos y negativos siempre con signo y relleno de espacios a la derecha.

Solución:

```
numeros = [5, -3, 12]
for n in numeros:
    print("{: >+5}".format(n))
```

Explicación:

- `+` → muestra siempre el signo
- `>5` → ancho total 5, alineado a la derecha

Ejercicio 34: Reporte de ventas con porcentaje

Enunciado:
Muestra ventas y porcentaje del total, formateando con 1 decimal.

Solución:

```
productos = ["A", "B", "C"]
ventas = [50, 30, 20]
total = sum(ventas)

for p, v in zip(productos, ventas):
    porcentaje = v / total
    print(f"{p}: {v} ventas ({porcentaje:.1%})")
```

Explicación:

- `:.1%` → convierte a porcentaje con 1 decimal
- f-string permite incluir variables y expresiones directamente

---

Ejercicio 35: Formato con relleno y centrado

Enunciado:
Pide un texto y muéstralo centrado en 30 caracteres, rellenando con `-`.

Solución:

```
texto = input("Texto: ")
print("{:-^30}".format(texto))
```

Explicación:

- `-^30` → rellena con `-` y centra el texto en ancho 30

Ejercicio 36: Mostrar números alineados en columnas

Enunciado:
Imprime los números del 1 al 10 y su doble en columnas alineadas.

Solución:

```
for i in range(1, 11):
    print("{:<3} {:>3}".format(i, i*2))
```

Explicación:

- `<3` → ancho 3, alineado a la izquierda
- `>3` → ancho 3, alineado a la derecha

Ejercicio 37: Combinación de texto y número con relleno

Enunciado:
Muestra un nombre y un número con relleno de `.` para alinear a la derecha.

Solución:

```
nombre = input("Nombre: ")
numero = int(input("Número: "))
print("{:.<20}{:>5}".format(nombre, numero))
```

Explicación:

- `. <20` → rellena con puntos hasta ancho 20
- `>5` → número alineado a la derecha

Ejercicio 38: Tabla con f-strings y expresiones

Enunciado:
Crea una tabla con números y sus cuadrados y cubos usando f-strings.

Solución:

```
for i in range(1, 6):
    print(f"{i:<3} {i**2:<4} {i**3:<5}")
```

Explicación:

- `<3, <4, <5` → ancho y alineación a la izquierda
- `{i**2}` y `{i**3}` → expresiones dentro de f-string

Ejercicio 39: Mostrar número en hexadecimal y binario

Enunciado:
Pide un número entero y muéstralo en decimal, hexadecimal y binario alineados.

Solución:

```
numero = int(input("Número entero: "))
print("{:<5} {:<5} {:<8}".format(numero, hex(numero)[2:], bin(numero)[2:]))
```

Explicación:

- `hex(numero)[2:]` → hexadecimal sin prefijo `0x`
- `bin(numero)[2:]` → binario sin `0b`
- `<5, <8` → alineación en columnas

Ejercicio 40: Tabla con encabezados y relleno

Enunciado:
Crea una tabla de estudiantes, notas y porcentaje usando relleno de espacios y decimales.

Solución:

```
estudiantes = ["Ana", "Luis", "Eva"]
notas = [8.5, 9.0, 7.5]
total = sum(notas)

print("{:<10} {:>6} {:>8}".format("Estudiante", "Nota", "% del total"))
for e, n in zip(estudiantes, notas):
    porcentaje = n / total
    print("{:<10} {:>6.1f} {:>8.2%}".format(e, n, porcentaje))
```

Explicación:

- `<10` → nombre alineado a la izquierda
- `>6.1f` → nota con 1 decimal alineada a la derecha
- `>8.2%` → porcentaje con 2 decimales, alineado a la derecha

Ejercicio 41: Formato de moneda

Enunciado:
Pide un precio y muéstralo como moneda con 2 decimales y el símbolo `$`.

Solución:

```
precio = float(input("Precio: "))
print("Precio: ${:.2f}".format(precio))
```

Explicación:

- `:.2f` → 2 decimales
- `$` → se muestra directamente como símbolo de moneda

Ejercicio 42: Tabla de productos con moneda

Enunciado:
Crea una tabla de productos y precios en formato de moneda alineados.

Solución:

```
productos = ["Leche", "Pan", "Queso"]
precios = [1.5, 0.8, 2.3]

print("{:<10} {:>10}".format("Producto", "Precio"))
for p, pr in zip(productos, precios):
    print("{:<10} ${:>8.2f}".format(p, pr))
```

Explicación:

- `<10` → producto alineado a la izquierda
- `>8.2f` → precio alineado a la derecha con 2 decimales

Ejercicio 43: Tabla con relleno de guiones

Enunciado:
Muestra nombres y precios con guiones como separadores hasta 20 caracteres.

Solución:

```
productos = ["Leche", "Pan", "Queso"]
precios = [1.5, 0.8, 2.3]

for p, pr in zip(productos, precios):
    print("{:-<15} ${:>5.2f}".format(p, pr))
```

Explicación:

- `-<15` → rellena con guiones hasta 15 caracteres
- `>5.2f` → precio alineado a la derecha con 2 decimales

Ejercicio 44: Tabla con f-string y porcentaje

Enunciado:
Muestra ventas de productos y porcentaje del total usando f-strings.

Solución:

```
productos = ["A", "B", "C"]
ventas = [50, 30, 20]
total = sum(ventas)

for p, v in zip(productos, ventas):
    print(f"{p:<5} {v:>5} ({v/total:.1%})")
```

Explicación:

- `<5` → producto alineado a la izquierda
- `>5` → ventas alineadas a la derecha
- `:.1%` → porcentaje con 1 decimal

Ejercicio 45: Tabla de calificaciones con promedio

Enunciado:
Muestra estudiantes, 3 notas y el promedio con decimales alineados.

Solución:

```
estudiantes = ["Ana", "Luis", "Eva"]
notas = [[8, 9, 7], [7, 8, 9], [10, 9, 8]]


print("{:<10} {:>5} {:>5} {:>5} {:>7}".format("Estudiante", "N1", "N2", "N3", "Prom"))
for e, n in zip(estudiantes, notas):
    promedio = sum(n)/3
    print("{:<10} {:>5} {:>5} {:>5} {:>7.2f}".format(e, n[0], n[1], n[2], promedio))
```

Explicación:

- `<10` → estudiante alineado a la izquierda
- `>5` → notas alineadas a la derecha
- `>7.2f` → promedio con 2 decimales

Ejercicio 46: Tabla de inventario con relleno

Enunciado:
Imprime productos, cantidades y precios con relleno de puntos para alinear.

Solución:

```
productos = ["Leche", "Pan", "Queso"]
cantidades = [5, 10, 7]
precios = [1.5, 0.8, 2.3]

for p, c, pr in zip(productos, cantidades, precios):
    print("{:.<10} {:>3} ${:>6.2f}".format(p, c, pr))
```

Explicación:

- `. <10` → rellena con puntos hasta 10 caracteres
- `>3` → cantidad alineada a la derecha
- `>6.2f` → precio con 2 decimales alineado a la derecha

Ejercicio 47: Formato mixto con f-string

Enunciado:
Pide nombre, edad y altura y muéstralos en una frase usando f-string con 1 decimal en altura.

Solución:

```
nombre = input("Nombre: ")
edad = int(input("Edad: "))
altura = float(input("Altura (m): "))
print(f"{nombre} tiene {edad} años y mide {altura:.1f} m")
```

Explicación:

- `:.1f` → altura con 1 decimal
- f-string permite mezclar texto y variables fácilmente

Ejercicio 48: Alinear y centrar títulos

Enunciado:
Muestra un título centrado con ancho 40 y relleno de `*`.

Solución:

```
titulo = input("Título: ")
print("{:*^40}".format(titulo))
```

Explicación:

- `*^40` → centra el texto en 40 caracteres con `*` de relleno

Ejercicio 49: Tabla con números y porcentajes

Enunciado:
Muestra productos, ventas y porcentaje del total en columnas alineadas.

Solución:

```
productos = ["A", "B", "C"]
ventas = [50, 30, 20]
total = sum(ventas)

print("{:<10} {:>6} {:>8}".format("Producto", "Ventas", "% Total"))
for p, v in zip(productos, ventas):
    print("{:<10} {:>6} {:>7.2%}".format(p, v, v/total))
```

Explicación:

- `<10` → producto alineado a la izquierda
- `>6` → ventas alineadas a la derecha
- `>7.2%` → porcentaje con 2 decimales

Ejercicio 50: Tabla completa de inventario

Enunciado:
Imprime productos, cantidad, precio unitario y total (cantidad × precio) alineados y formateados como moneda.

Solución:

```
productos = ["Leche", "Pan", "Queso"]
cantidades = [5, 10, 7]
precios = [1.5, 0.8, 2.3]


print("{:<10} {:>8} {:>10} {:>10}".format("Producto", "Cantidad", "Precio", "Total"))
for p, c, pr in zip(productos, cantidades, precios):
    total = c * pr
    print("{:<10} {:>8} ${:>9.2f} ${:>9.2f}".format(p, c, pr, total))
```

Explicación:

- `<10` → producto alineado a la izquierda
- `>8` → cantidad alineada a la derecha
- `>9.2f` → precio y total con 2 decimales, alineados a la derecha
- Multiplicación `c*pr` calcula total por producto

MANUAL DE REFERENCIA: GRÁFICOS EN PYTHON (MATPLOTLIB)

(Nota: Todos los ejercicios asumen que la librería Matplotlib se ha importado inicialmente con el alias `plt`: `import matplotlib.pyplot as plt`. Los ejercicios 35, 36, 41 y 47 requieren la importación de `Axes3D`.)

EJERCICIOS BÁSICOS DE VISUALIZACIÓN (1-10)

Ejercicio	Enunciado y Solución	Comandos y Controles Explicados
1: Gráfico de línea simple	Muestra los números del 1 al 5 en el eje X y sus cuadrados en el eje Y. `plt.plot(x, y)` `plt.show()`	**`import matplotlib.pyplot as plt`**: Importa la librería con el alias `plt`. **`x`**: Valores en el eje X. **`y`**: Valores en el eje Y (calculando el cuadrado de cada X). **`plt.plot(x, y)`**: Crea un gráfico de línea. **`plt.show()`**: Muestra el gráfico en pantalla/abre una ventana con el gráfico.
2: Gráfico de puntos (scatter)	Muestra los puntos (1,2), (2,4), (3,6) usando un gráfico de dispersión. `plt.scatter(x, y)` `plt.show()`	**`plt.scatter()`**: Dibuja los puntos como símbolos en las coordenadas dadas.
3: Gráfico de barras	Crea un gráfico de barras con productos y sus ventas. `plt.bar(productos, ventas)` `plt.show()`	**`plt.bar()`**: Dibuja barras verticales, cuya altura está determinada por las ventas.
4: Gráfico de barras horizontales	Crea un gráfico de barras horizontal para los mismos datos. `plt.barh(productos, ventas)` `plt.show()`	**`plt.barh()`**: Dibuja barras horizontales.
5: Título y etiquetas	Agrega título y etiquetas a los ejes en un gráfico de línea. `plt.title(...)` `plt.xlabel(...)` `plt.ylabel(...)`	**`plt.title()`**: Agrega un título al gráfico. **`plt.xlabel()`**: Etiqueta del eje X. **`plt.ylabel()`**: Etiqueta del eje Y.
6: Cambiar color y estilo de línea	Haz que la línea sea roja y punteada. `plt.plot(x, y, 'r--')`	**`'r--'`**: Define la línea roja (`r`) y punteada (`--`).
7: Añadir puntos en la línea	Dibuja una línea azul y muestra los puntos con círculos. `plt.plot(x, y, 'bo-')`	**`'b'`**: Define el color azul. **`'o'`**: Utiliza un marcador circular en cada punto. **`'-'`**: Dibuja una línea conectando los puntos.
8: Gráfico de pastel	Muestra la proporción de ventas de los productos como un gráfico circular. `plt.pie(ventas, labels=productos)`	**`plt.pie()`**: Crea un gráfico de pastel. **`labels`**: Define los nombres de cada sector.
9: Leyenda	Dibuja dos líneas y agrega	**`label`**: Define el nombre de la línea que se usará

	leyenda. `plt.plot(..., label="Cuadrado") plt.legend()`	para la leyenda. **`plt.legend()`**: Muestra la leyenda en el gráfico.
10: Ajustar límites del gráfico	Cambia los límites de los ejes X e Y. `plt.xlim(0, 6)` `plt.ylim(0, 30)`	**`plt.xlim(a, b)`**: Establece el rango del eje X. **`plt.ylim(a, b)`**: Establece el rango del eje Y.

EJERCICIOS DE ESTILOS Y MÚLTIPLES SERIES (11-20)

Ejercicio	Enunciado y Solución	Comandos y Controles Explicados
11: Múltiples líneas	Dibuja dos líneas en el mismo gráfico: $y = x^2$ y $y = x^3$. `plt.plot(x, y1, 'r-', label="x^2")` `plt.plot(x, y2, 'b--', label="x^3")`	**`'r-'`**: Línea roja continua. **`'b--'`**: Línea azul punteada. Se utiliza `plt.plot()` varias veces para superponer las líneas.
12: Barras con colores diferentes	Muestra barras de productos con diferentes colores. `plt.bar(productos, ventas, color=colores)`	**`color=colores`**: Asigna un color diferente a cada barra vertical, usando una lista de colores predefinida.
13: Subplots (vertical)	Muestra dos gráficos de línea en la misma figura: uno arriba y otro abajo. `plt.subplot(2,1,1)` `plt.subplot(2,1,2)` `plt.tight_layout()`	**`plt.subplot(filas, columnas, índice)`**: Divide la figura en subgráficos (aquí 2 filas, 1 columna). **`plt.tight_layout()`**: Ajusta los espacios entre los subgráficos automáticamente.
14: Gráfico de dispersión con color	Dibuja un gráfico de dispersión con puntos de diferentes colores. `plt.scatter(x, y, c=colores)`	**`c=colores`**: Asigna un color a cada punto de dispersión, tomando los valores de la lista `colores`.
15: Barras horizontales con colores	Muestra un gráfico de barras horizontal con colores personalizados. `plt.barh(productos, ventas, color=['cyan', 'magenta', 'yellow'])`	**`plt.barh()`**: Dibuja las barras en horizontal. **`color=[...]`**: Especifica los colores para cada barra.
16: Añadir cuadrícula	Dibuja una línea y agrega cuadrícula. `plt.grid(True)`	**`plt.grid(True)`**: Activa la visualización de la cuadrícula en el gráfico.
17: Cambiar grosor de línea	Dibuja una línea roja más gruesa. `plt.plot(x, y, 'r-', linewidth=3)`	**`linewidth=3`**: Define el grosor de la línea.
18: Gráfico de pastel con porcentaje	Muestra ventas como un gráfico de pastel con porcentajes. `plt.pie(..., autopct='%1.1f%%')`	**`autopct='%1.1f%%'`**: Muestra el porcentaje de cada sector con un decimal.
19: Exploding pie (resaltar sector)	Resalta el producto con mayor venta en el gráfico de pastel. `explode = [0.1, 0, 0]` `plt.pie(..., explode=explode, ...)`	**`explode`**: Define una lista de desplazamiento que separa un sector del pastel para resaltarlo.
20: Guardar gráfico en	Crea un gráfico de línea y guárdalo como imagen PNG.	**`plt.savefig("archivo.png")`**: Guarda el gráfico como una imagen PNG

archivo	`plt.savefig("grafico_linea.png")`	(u otro formato) antes de que se muestre en pantalla.

EJERCICIOS DE FORMATO AVANZADO (21-30)

Ejercicio	Enunciado y Solución	Comandos y Controles Explicados
21: Subplots lado a lado	Muestra dos gráficos uno al lado del otro: línea y barras. `plt.subplot(1, 2, 1)` `plt.subplot(1, 2, 2)`	**`subplot(1, 2, 1)`**: Indica 1 fila, 2 columnas, y selecciona el primer gráfico. **`subplot(1, 2, 2)`**: Selecciona el segundo gráfico. **`plt.tight_layout()`**: Ajusta espacios automáticamente.
22: Scatter con tamaño variable	Dibuja un gráfico de dispersión donde el tamaño de los puntos depende de otro valor. `sizes = [...]` `plt.scatter(x, y, s=sizes)`	**`s=sizes`**: Define el tamaño de cada punto en el gráfico de dispersión (scatter).
23: Scatter con color y tamaño	Agrega colores a los puntos además de tamaño variable. `plt.scatter(x, y, s=sizes, c=colors)`	**`c=colors`**: Asigna un color específico a cada punto. **`s=sizes`**: Asigna el tamaño a cada punto.
24: Barra con etiquetas encima	Muestra valores encima de cada barra. `for i, v in enumerate(ventas):` `plt.text(i, v + 0.5, str(v), ha='center')`	**`plt.text(x, y, texto)`**: Escribe texto en las coordenadas especificadas (x, y). **`ha='center'`**: Centra el texto horizontalmente sobre el punto (x, y).
25: Línea con estilo y grosor	Dibuja una línea verde punteada más gruesa. `plt.plot(x, y, 'g--', linewidth=4)`	**`'g--'`**: Define la línea como verde punteada. **`linewidth=4`**: Aumenta el grosor de la línea.
26: Barras con color según valor	Colorea las barras según si la venta es mayor o menor a 10. `colors = ['red' if v>10 else 'blue' for v in ventas]` `plt.bar(productos, ventas, color=colors)`	Se utiliza una **lista de comprensión** para asignar el color (rojo o azul) basándose en una condición.
27: Pie con porcentaje y sombra	Muestra un gráfico circular con porcentaje y sombra. `plt.pie(..., shadow=True)`	**`shadow=True`**: Agrega una sombra al gráfico de pastel.
28: Línea con marcador y color personalizado	Dibuja una línea con marcador de estrella y color naranja. `plt.plot(x, y, marker='*', color='orange')`	**`marker='*'`**: Define la forma del marcador a usar en cada punto. **`color='orange'`**: Define el color de la línea.
29: Subplots con tres gráficos	Crea una figura con tres gráficos de línea en vertical. `plt.subplot(3,1,indice)`	**`subplot(3,1,indice)`**: Configura la figura con 3 filas y 1 columna. Permite usar diferentes estilos de línea para cada subgráfico (`'r-'`, `'b--'`, `'g.-'`).
30: Gráfico de barras con borde	Dibuja barras con borde negro. `plt.bar(..., color='skyblue',`	**`edgecolor='black'`**: Define el color del borde de las barras. **`color='skyblue'`**: Define el color

	`edgecolor='black')`	del interior de las barras.

EJERCICIOS DE DISTRIBUCIÓN Y ÁREA (31-40)

Ejercicio	Enunciado y Solución	Comandos y Controles Explicados
31: Histograma simple	Muestra la distribución de edades usando un histograma. `plt.hist(edades, bins=4, ...)`	**`plt.hist()`**: Dibuja un histograma. **`bins=4`**: Establece el número de intervalos o "contenedores" en el histograma. **`edgecolor='black'`**: Añade borde a cada barra.
32: Histogramas múltiples	Compara dos conjuntos de edades en un mismo histograma. `plt.hist([edades1, edades2], ...)`	Se pasa una lista de listas a `plt.hist()` para generar un histograma múltiple. **`label`**: Se usa para nombrar cada grupo en la leyenda.
33: Gráfico de área apilada (stack plot)	Muestra ventas acumuladas de dos productos a lo largo de días. `plt.stackplot(dias, prod1, prod2, labels=..., colors=...)`	**`plt.stackplot()`**: Crea gráficos de área apilados. **`labels`** y **`colors`**: Permiten definir la leyenda y los colores de las áreas.
34: Líneas con anotaciones	Añade anotación al punto máximo. `plt.annotate('Máximo', xy=(4,8), xytext=(3,9), arrowprops=...)`	**`plt.annotate()`**: Agrega texto con una flecha. **`xy`**: Coordenadas del punto al que apunta la flecha. **`xytext`**: Posición donde se ubicará el texto de la anotación. **`arrowprops`**: Propiedades de la flecha.
35: Gráfico 3D de líneas	Dibuja una línea en 3D con coordenadas (x, y, z). `fig.add_subplot(111, projection='3d')` `ax.plot(x, y, z, color='red')`	**`from mpl_toolkits.mplot3d import Axes3D`**: Importación requerida para gráficos 3D. **`projection='3d'`**: Activa la representación tridimensional del gráfico. **`ax.plot(x, y, z)`**: Dibuja la línea en el espacio 3D.
36: Gráfico 3D de puntos	Muestra puntos 3D con color y tamaño. `ax.scatter(x, y, z, c='green', s=50)`	**`ax.scatter()`**: Dibuja puntos en 3D. **`c`**: Define el color de los puntos. **`s`**: Define el tamaño de los puntos.
37: Líneas con estilo personalizado	Dibuja línea azul con marcador cuadrado y línea discontinua. `plt.plot(..., linestyle='--', marker='s', color='blue')`	**`linestyle='--'`**: Establece la línea como discontinua. **`marker='s'`**: Define el marcador como un cuadrado.
38: Gráfico de barras apiladas	Muestra ventas acumuladas de dos productos en barras apiladas. `plt.bar(x, prod1, ...)` `plt.bar(x, prod2, bottom=prod1, ...)`	**`bottom=prod1`**: Comando clave que indica que la segunda barra (`prod2`) debe apilarse (empezar) sobre el final de la primera barra (`prod1`).
39: Pastel con separación y sombra	Resalta el mayor producto y agrega sombra. `plt.pie(..., explode=explode, shadow=True, autopct='...')`	Combina los controles de resaltado (`explode`), sombra (`shadow=True`) y porcentaje (`autopct`).

40: Gráfico combinado	Muestra barras y línea sobre el mismo gráfico. `plt.bar(x, y, ...)` `plt.plot(x, y2, ...)`	Demuestra la combinación de las funciones `plt.bar()` y `plt.plot()` en una sola figura, utilizando `label` y `plt.legend()` para diferenciar las series.

EJERCICIOS AVANZADOS Y ESPECIALES (41-50)

Ejercicio	Enunciado y Solución	Comandos y Controles Explicados
41: Gráfico 3D de barras	Crea un gráfico 3D de barras. `ax.bar3d(x, y, z, dx, dy, dz, color='cyan')`	**`import numpy as np`**: Requerido para manejar las estructuras de datos 3D. **`ax.bar3d()`**: Crea las barras en 3D. **`dx, dy`**: Definen el tamaño de la base de la barra en X e Y. **`dz`**: Define la altura de cada barra.
42: Líneas con múltiples estilos	Dibuja tres líneas con diferente estilo, color y marcador. `plt.plot(x, y1, 'r-o', ...)` `plt.plot(x, y2, 'b--s', ...)` `plt.plot(x, y3, 'g-.^', ...)`	**`'r-o'`**: Línea roja continua con marcador de círculo. **`'b--s'`**: Línea azul punteada con marcador cuadrado. **`'g-.^'`**: Línea verde de puntos y rayas con marcador triángulo.
44: Stack plot avanzado	Muestra acumulado de tres productos. `plt.stackplot(dias, prod1, prod2, prod3, ...)`	Utiliza `plt.stackplot()` para graficar tres series apiladas (`prod1`, `prod2`, `prod3`), controlando la leyenda y los colores.
43: Subplots complejos (2x2)	Crea 4 subplots en 2x2 con diferentes tipos de gráficos. `fig, axs = plt.subplots(2,2)` `axs[i,j].plot(...)`	**`plt.subplots(2,2)`**: Crea una figura con una cuadrícula de 2 filas por 2 columnas. **`axs[i,j]`**: Permite acceder y manipular cada gráfico individualmente (ej. `axs` es el gráfico superior izquierdo). **`set_title()`**: Define el título para un subplot específico.
45: Línea con sombreado	Dibuja línea y rellena área debajo de la curva. `plt.fill_between(x, y1, color='blue', alpha=0.2)`	**`fill_between(x, y)`**: Rellena el área bajo la línea definida por (x, y). **`alpha`**: Define la transparencia del color de relleno (0.2 en este caso).
46: Barras con anotaciones y colores	Muestra barras y escribe el valor encima con color personalizado. `plt.text(x[i], v+0.5, str(v), ha='center', color='red')`	Combina la creación de barras con el comando **`plt.text()`**, especificando el `color` del texto de la anotación (ej. `'red'`).
47: Gráfico 3D de superficie	Crea un gráfico 3D de superficie usando malla de puntos X y Y. `X, Y = np.meshgrid(X, Y)` `ax.plot_surface(X, Y, Z, cmap='viridis')`	**`np.meshgrid()`**: Crea una malla de coordenadas X e Y, fundamental para generar superficies 3D. **`ax.plot_surface()`**: Dibuja la superficie 3D. **`cmap='viridis'`**: Define el mapa de colores utilizado en la superficie.
48: Gráfico	Crea gráfico de barras y línea compartiendo	**`ax1.twinx()`**: Crea un segundo eje

combinado con secundario	eje X, pero línea con eje Y derecho. `fig, ax1 = plt.subplots()` `ax2 = ax1.twinx()`	Y (`ax2`) que comparte el mismo eje X que el eje primario (`ax1`). Esto es crucial para mostrar escalas de datos muy diferentes en un solo gráfico.
49: Gráfico de dispersión con gradiente	Dibuja scatter con color proporcional a los valores de Y. `plt.scatter(x, y2, c=y2, cmap='plasma', s=100)` `plt.colorbar()`	**`c=y2`**: Hace que el color de cada punto dependa de su valor en la variable `y2`. **`cmap='plasma'`**: Define el mapa de colores a utilizar para el gradiente. **`plt.colorbar()`**: Muestra la escala de colores que indica la relación entre el color y el valor de los datos.
50: Animación básica	Crea animación de un punto moviéndose en línea. `import matplotlib.animation as animation` `ani = animation.FuncAnimation(fig, animate, frames=6, interval=500, blit=True)`	**`FuncAnimation()`**: Comando principal para crear la animación. **`animate(i)`**: Es la función que se ejecuta repetidamente para actualizar los datos de la gráfica en cada paso. **`frames`**: Define el número de pasos o cuadros de la animación. **`interval`**: Define el tiempo (en milisegundos) entre cada cuadro. **`blit=True`**: Opción que optimiza la animación para un redibujado más rápido.

Ejercicio 1: Gráfico de línea simple

Enunciado:
Crea un gráfico de línea mostrando los números del 1 al 5 en el eje X y sus cuadrados en el eje Y.

Solución:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [i**2 for i in x]

plt.plot(x, y)
plt.show()
```

Explicación:

- `x` → valores en eje X
- `y` → valores en eje Y, calculando el cuadrado de cada X
- `plt.plot()` → crea un gráfico de línea
- `plt.show()` → muestra el gráfico en pantalla

Comandos explicados:

- `import matplotlib.pyplot as plt` → importa la librería Matplotlib con alias `plt`
- `plt.plot(x, y)` → dibuja una línea conectando los puntos `(x[i], y[i])`
- `plt.show()` → abre una ventana con el gráfico

Ejercicio 2: Gráfico de puntos (scatter)

Enunciado:
Muestra los puntos `(1,2)`, `(2,4)`, `(3,6)` usando un gráfico de dispersión.

Solución:

```
x = [1, 2, 3]
y = [2, 4, 6]

plt.scatter(x, y)
plt.show()
```

Explicación:

- `plt.scatter()` dibuja los puntos como símbolos en las coordenadas dadas

---

Ejercicio 3: Gráfico de barras

Enunciado:
Crea un gráfico de barras con productos y sus ventas.

Solución:

```
productos = ["A", "B", "C"]
ventas = [10, 15, 7]

plt.bar(productos, ventas)
plt.show()
```

Explicación:

- `plt.bar()` dibuja barras verticales con altura según `ventas`

Ejercicio 4: Gráfico de barras horizontales

Enunciado:
Crea un gráfico de barras horizontal para los mismos datos.

Solución:

```
plt.barh(productos, ventas)
plt.show()
```

Explicación:

- `plt.barh()` dibuja barras horizontales

Ejercicio 5: Título y etiquetas

Enunciado:
Agrega título y etiquetas a los ejes en un gráfico de línea.

Solución:

```
plt.plot(x, y)
plt.title("Gráfico de cuadrados")
plt.xlabel("Valores X")
plt.ylabel("Valores Y")
plt.show()
```

Explicación:

- `plt.title()` → agrega título al gráfico
- `plt.xlabel()` → etiqueta del eje X
- `plt.ylabel()` → etiqueta del eje Y

Ejercicio 6: Cambiar color y estilo de línea

Enunciado:
Haz que la línea sea roja y punteada.

Solución:

```
plt.plot(x, y, 'r--')
plt.show()
```

Explicación:

- `'r--'` → línea roja (`r`) y punteada (`--`)

Ejercicio 7: Añadir puntos en la línea

Enunciado:
Dibuja una línea azul y muestra los puntos con círculos.

Solución:

```
plt.plot(x, y, 'bo-')
plt.show()
```

Explicación:

- `'b'` → color azul
- `'o'` → marcador circular en cada punto
- `'-'` → línea conectando los puntos

Ejercicio 8: Gráfico de pastel

Enunciado:
Muestra la proporción de ventas de los productos como un gráfico circular.

Solución:

```
plt.pie(ventas, labels=productos)
plt.show()
```

Explicación:

- `plt.pie()` → crea gráfico de pastel
- `labels` → nombres de cada sector

Ejercicio 9: Leyenda

Enunciado:
Dibuja dos líneas y agrega leyenda.

Solución:

```
y2 = [i*2 for i in x]
plt.plot(x, y, label="Cuadrado")
plt.plot(x, y2, label="Doble")
plt.legend()
plt.show()
```

Explicación:

- `label` → nombre de la línea para la leyenda
- `plt.legend()` → muestra la leyenda en el gráfico

Ejercicio 10: Ajustar límites del gráfico

Enunciado:
Cambia los límites de los ejes X e Y.

Solución:

```
plt.plot(x, y)
plt.xlim(0, 6)
plt.ylim(0, 30)
plt.show()
```

Explicación:

- `plt.xlim(a, b)` → rango del eje X
- `plt.ylim(a, b)` → rango del eje Y

Ejercicio 11: Múltiples líneas en un gráfico

Enunciado:
Dibuja dos líneas en el mismo gráfico: `y = x^2` y `y = x^3`.

Solución:

```
x = [1, 2, 3, 4, 5]
y1 = [i**2 for i in x]
y2 = [i**3 for i in x]

plt.plot(x, y1, 'r-', label="x^2")
plt.plot(x, y2, 'b--', label="x^3")
plt.legend()
plt.show()
```

Explicación:

- `'r-'` → línea roja continua
- `'b--'` → línea azul punteada
- `plt.legend()` → muestra leyenda

Ejercicio 12: Gráfico de barras con colores diferentes

Enunciado:

Muestra barras de productos con diferentes colores.

Solución:

```
productos = ["A", "B", "C"]
ventas = [10, 15, 7]
colores = ['red', 'green', 'blue']

plt.bar(productos, ventas, color=colores)
plt.show()
```

Explicación:

- `color=colores` → asigna un color a cada barra

Ejercicio 13: Subplots (dos gráficos en una ventana)

Enunciado:
Muestra dos gráficos de línea en la misma figura: uno arriba y otro abajo.

Solución:

```
x = [1, 2, 3, 4, 5]
y1 = [i for i in x]
y2 = [i**2 for i in x]

plt.subplot(2,1,1)
plt.plot(x, y1, 'g-o')
plt.title("y = x")

plt.subplot(2,1,2)
plt.plot(x, y2, 'b--')
plt.title("y = x^2")

plt.tight_layout()
plt.show()
```

Explicación:

- `plt.subplot(filas, columnas, índice)` → divide la figura en subgráficos
- `plt.tight_layout()` → ajusta espacios automáticamente

Ejercicio 14: Gráfico de dispersión con color

Enunciado:
Dibuja un gráfico de dispersión con puntos de diferentes colores.

Solución:

```
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
colores = ['red', 'blue', 'green', 'purple', 'orange']

plt.scatter(x, y, c=colores)
plt.show()
```

Explicación:

- `c=colores` → asigna color a cada punto

Ejercicio 15: Gráfico de barras horizontal con colores

Enunciado:
Muestra un gráfico de barras horizontal con colores personalizados.

Solución:

```
plt.barh(productos, ventas, color=['cyan', 'magenta', 'yellow'])
plt.show()
```

Explicación:

- `plt.barh()` → barras horizontales
- `color=[...]` → colores para cada barra

Ejercicio 16: Añadir cuadrícula

Enunciado:
Dibuja una línea y agrega cuadrícula.

Solución:

```
plt.plot(x, y)
plt.grid(True)
plt.show()
```

Explicación:

- `plt.grid(True)` → activa la cuadrícula

Ejercicio 17: Cambiar grosor de línea

Enunciado:
Dibuja una línea roja más gruesa.

Solución:

```
plt.plot(x, y, 'r-', linewidth=3)
plt.show()
```

Explicación:

- `linewidth=3` → grosor de la línea

Ejercicio 18: Gráfico de pastel con porcentaje

Enunciado:
Muestra ventas como un gráfico de pastel con porcentajes.

Solución:

```
plt.pie(ventas, labels=productos, autopct='%1.1f%%')
plt.show()
```

Explicación:

- `autopct='%1.1f%%'` → muestra porcentaje con 1 decimal en cada sector

Ejercicio 19: Exploding pie (resaltar sector)

Enunciado:

Resalta el producto con mayor venta en el gráfico de pastel.

Solución:

```
explode = [0.1, 0, 0]  # resalta la primera barra
plt.pie(ventas, labels=productos, explode=explode, autopct='%1.1f%%')
plt.show()
```

Explicación:

- `explode` → desplaza sectores del pastel para resaltar

Ejercicio 20: Guardar gráfico en archivo

Enunciado:
Crea un gráfico de línea y guárdalo como imagen PNG.

Solución:

```
plt.plot(x, y)
plt.title("Línea")
plt.savefig("grafico_linea.png")
plt.show()
```

Explicación:

- `plt.savefig("archivo.png")` → guarda el gráfico como imagen antes de mostrarlo

Ejercicio 21: Subplots lado a lado

Enunciado:
Muestra dos gráficos uno al lado del otro: línea y barras.

Solución:

```
x = [1, 2, 3, 4, 5]
y = [i**2 for i in x]

plt.subplot(1, 2, 1)
plt.plot(x, y, 'r-o')
plt.title("Línea")

plt.subplot(1, 2, 2)
plt.bar(x, y, color='cyan')
plt.title("Barras")

plt.tight_layout()
plt.show()
```

Explicación:

- `subplot(1,2,1)` → 1 fila, 2 columnas, primer gráfico
- `subplot(1,2,2)` → segundo gráfico
- `tight_layout()` → ajusta espacios automáticamente

Ejercicio 22: Scatter con tamaño variable

Enunciado:
Dibuja un gráfico de dispersión donde el tamaño de los puntos depende de otro valor.

Solución:

```
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
sizes = [50, 100, 150, 200, 250]


plt.scatter(x, y, s=sizes)
plt.show()
```

Explicación:

- `s=sizes` → tamaño de cada punto

Ejercicio 23: Scatter con color y tamaño

Enunciado:
Agrega colores a los puntos además de tamaño variable.

Solución:

```
colors = ['red', 'blue', 'green', 'purple', 'orange']
plt.scatter(x, y, s=sizes, c=colors)
plt.show()
```

Explicación:

- `c=colors` → asigna un color a cada punto

Ejercicio 24: Barra con etiquetas encima

Enunciado:
Muestra valores encima de cada barra.

Solución:

```
productos = ["A", "B", "C"]
ventas = [10, 15, 7]


plt.bar(productos, ventas)
for i, v in enumerate(ventas):
    plt.text(i, v + 0.5, str(v), ha='center')
plt.show()
```

Explicación:

- `plt.text(x, y, texto)` → escribe texto en coordenadas `(x, y)`
- `ha='center'` → centra el texto horizontalmente

Ejercicio 25: Gráfico de línea con estilo y grosor

Enunciado:
Dibuja una línea verde punteada más gruesa.

Solución:

```
plt.plot(x, y, 'g--', linewidth=4)
plt.show()
```

Explicación:

- `'g--'` → línea verde punteada
- `linewidth=4` → grosor

Ejercicio 26: Gráfico de barras con color según valor

Enunciado:
Colorea las barras según si la venta es mayor o menor a 10.

Solución:

```
colors = ['red' if v>10 else 'blue' for v in ventas]
plt.bar(productos, ventas, color=colors)
plt.show()
```

Explicación:

- Lista de comprensión para asignar color según condición

Ejercicio 27: Pie con porcentaje y sombra

Enunciado:
Muestra un gráfico circular con porcentaje y sombra.

Solución:

```
plt.pie(ventas, labels=productos, autopct='%1.1f%%', shadow=True)
plt.show()
```

Explicación:

- `shadow=True` → agrega sombra al gráfico de pastel

Ejercicio 28: Línea con marcador y color personalizado

Enunciado:
Dibuja una línea con marcador de estrella y color naranja.

Solución:

```
plt.plot(x, y, marker='*', color='orange')
plt.show()
```

Explicación:

- `marker='*'` → forma del marcador
- `color='orange'` → color de la línea

Ejercicio 29: Subplots con tres gráficos

Enunciado:
Crea una figura con tres gráficos de línea en vertical.

Solución:

```
y2 = [i*2 for i in x]
y3 = [i*3 for i in x]


plt.subplot(3,1,1)
plt.plot(x, y, 'r-')
plt.title("x^2")

plt.subplot(3,1,2)
plt.plot(x, y2, 'b--')
plt.title("x*2")

plt.subplot(3,1,3)
plt.plot(x, y3, 'g-.')
plt.title("x*3")

plt.tight_layout()
plt.show()
```

Explicación:

- `subplot(3,1,indice)` → 3 filas, 1 columna
- Diferentes estilos de línea para cada gráfico

Ejercicio 30: Gráfico de barras con borde

Enunciado:
Dibuja barras con borde negro.

Solución:

```
plt.bar(productos, ventas, color='skyblue', edgecolor='black')
plt.show()
```

Explicación:

- `edgecolor='black'` → color del borde de las barras
- `color='skyblue'` → color del interior de las barras

Ejercicio 31: Histograma simple

Enunciado:
Muestra la distribución de edades `[15, 16, 17, 15, 16, 18, 17]` usando un histograma.

Solución:

```
edades = [15, 16, 17, 15, 16, 18, 17]
plt.hist(edades, bins=4, color='skyblue', edgecolor='black')
plt.show()
```

Explicación:

- `plt.hist()` → dibuja histograma
- `bins=4` → número de intervalos
- `edgecolor='black'` → borde de cada barra

Ejercicio 32: Histogramas múltiples

Enunciado:
Compara dos conjuntos de edades en un mismo histograma.

Solución:

```
edades1 = [15, 16, 17, 15]
edades2 = [16, 17, 18, 16]


plt.hist([edades1, edades2], bins=4, color=['red','blue'],
label=['Grupo1','Grupo2'])
plt.legend()
plt.show()
```

Explicación:

- `plt.hist([lista1, lista2], ...)` → múltiple histograma
- `label` → nombres para leyenda

Ejercicio 33: Gráfico de área apilada (stack plot)

Enunciado:
Muestra ventas acumuladas de dos productos a lo largo de días.

Solución:

```
dias = [1, 2, 3, 4, 5]

prod1 = [3, 4, 2, 5, 3]

prod2 = [1, 2, 1, 3, 2]


plt.stackplot(dias, prod1, prod2, labels=['Prod1','Prod2'],
colors=['cyan','magenta'])

plt.legend()

plt.show()
```

Explicación:

- `plt.stackplot()` → gráficos apilados
- `labels` y `colors` → leyenda y colores

Ejercicio 34: Gráfico de líneas con anotaciones

Enunciado:
Añade anotación al punto máximo de `y=[2,4,6,8,7]`.

Solución:

```
y = [2,4,6,8,7]
x = [1,2,3,4,5]


plt.plot(x, y, 'b-o')
plt.annotate('Máximo', xy=(4,8), xytext=(3,9),
arrowprops=dict(facecolor='black'))
plt.show()
```

Explicación:

- `plt.annotate()` → agrega texto con flecha
- `xy` → coordenadas del punto
- `xytext` → posición del texto
- `arrowprops` → propiedades de la flecha

Ejercicio 35: Gráfico 3D de líneas

Enunciado:
Dibuja una línea en 3D con coordenadas `(x, y, z)`.

Solución:

```
from mpl_toolkits.mplot3d import Axes3D


fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')


x = [1,2,3,4,5]
y = [2,3,4,5,6]
z = [5,6,7,8,9]


ax.plot(x, y, z, color='red')
plt.show()
```

Explicación:

- `projection='3d'` → activa gráfico 3D
- `ax.plot(x, y, z)` → dibuja línea en 3D

Ejercicio 36: Gráfico 3D de puntos

Enunciado:
Muestra puntos 3D con color y tamaño.

Solución:

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(x, y, z, c='green', s=50)
plt.show()
```

Explicación:

- `ax.scatter()` → puntos en 3D
- `c` → color, `s` → tamaño

Ejercicio 37: Gráfico de líneas con estilo personalizado

Enunciado:
Dibuja línea azul con marcador cuadrado y línea discontinua.

Solución:

```
plt.plot(x, y, linestyle='--', marker='s', color='blue')
plt.show()
```

Explicación:

- `linestyle='--'` → línea discontinua
- `marker='s'` → marcador cuadrado
- `color='blue'` → color de la línea

Ejercicio 38: Gráfico de barras apiladas

Enunciado:
Muestra ventas acumuladas de dos productos en barras apiladas.

Solución:

```
plt.bar(x, prod1, color='cyan', label='Prod1')
plt.bar(x, prod2, bottom=prod1, color='magenta', label='Prod2')
plt.legend()
plt.show()
```

Explicación:

- `bottom=prod1` → apila la segunda barra sobre la primera

Ejercicio 39: Gráfico de pastel con separación y sombra

Enunciado:
Resalta el mayor producto y agrega sombra.

Solución:

```
ventas = [10, 15, 7]
productos = ["A","B","C"]
explode = [0,0.1,0]


plt.pie(ventas, labels=productos, explode=explode, shadow=True, autopct='%1.1f%%')
plt.show()
```

Explicación:

- `explode` → desplaza sector
- `shadow=True` → sombra
- `autopct` → porcentaje

Ejercicio 40: Gráfico combinado

Enunciado:
Muestra barras y línea sobre el mismo gráfico.

Solución:

```
plt.bar(x, y, color='lightblue', label='Barras')
plt.plot(x, y2, color='red', marker='o', label='Línea')
plt.legend()
plt.show()
```

Explicación:

- Combinación de barras y línea
- `label` y `plt.legend()` → leyenda

Ejercicio 41: Gráfico 3D de barras

Enunciado:
Crea un gráfico 3D de barras mostrando ventas por productos y días.

Solución:

```
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

x = np.array([1,2,3])
y = np.array([1,2,3])
z = np.zeros(3)
dx = dy = 0.5
dz = [5,7,3]

ax.bar3d(x, y, z, dx, dy, dz, color='cyan')
plt.show()
```

Explicación:

- `bar3d()` → crea barras en 3D
- `dx, dy` → tamaño en X e Y
- `dz` → altura de cada barra

Ejercicio 42: Gráfico de líneas con múltiples estilos

Enunciado:
Dibuja tres líneas con diferente estilo, color y marcador.

Solución:

```
x = [1,2,3,4,5]
y1 = [1,4,9,16,25]
y2 = [2,5,10,17,26]
y3 = [3,6,11,18,27]

plt.plot(x, y1, 'r-o', label='y1')
plt.plot(x, y2, 'b--s', label='y2')
plt.plot(x, y3, 'g-.^', label='y3')
plt.legend()
plt.show()
```

Explicación:

- `'r-o'` → línea roja con marcador círculo
- `'b--s'` → línea azul punteada con cuadrado
- `'g-.^'` → línea verde punteada y marcador triángulo

Ejercicio 43: Subplots complejos

Enunciado:
Crea 4 subplots en 2x2 con diferentes tipos de gráficos.

Solución:

```
fig, axs = plt.subplots(2,2)

axs[0,0].plot(x, y1, 'r')
axs[0,0].set_title("Línea")

axs[0,1].bar(x, y2, color='blue')
axs[0,1].set_title("Barras")

axs[1,0].scatter(x, y3, color='green')
axs[1,0].set_title("Dispersión")

axs[1,1].pie([10,20,30], labels=['A','B','C'], autopct='%1.1f%%')
axs[1,1].set_title("Pastel")

plt.tight_layout()
plt.show()
```

Explicación:

- `plt.subplots(2,2)` → crea 4 subplots en cuadrícula 2x2
- `axs[i,j]` → acceso a cada gráfico
- `set_title()` → título individual

Ejercicio 44: Stack plot avanzado

Enunciado:
Muestra acumulado de tres productos a lo largo de días con colores y leyenda.

Solución:

```
dias = [1,2,3,4,5]
prod1 = [3,4,2,5,3]
prod2 = [1,2,1,3,2]
prod3 = [2,1,3,2,1]

plt.stackplot(dias, prod1, prod2, prod3, labels=['Prod1','Prod2','Prod3'],
colors=['cyan','magenta','yellow'])
plt.legend()
plt.show()
```

Explicación:

- `plt.stackplot()` → gráfico de área apilada
- `labels` y `colors` → leyenda y colores

Ejercicio 45: Línea con sombreado

Enunciado:
Dibuja línea y rellena área debajo de la curva.

Solución:

```
plt.plot(x, y1, 'b')
plt.fill_between(x, y1, color='blue', alpha=0.2)
plt.show()
```

Explicación:

- `fill_between(x, y)` → rellena área bajo la línea
- `alpha` → transparencia

Ejercicio 46: Gráfico de barras con anotaciones y colores

Enunciado:
Muestra barras y escribe el valor encima con color personalizado.

Solución:

```
for i, v in enumerate(y2):
    plt.bar(x[i], v, color='orange')
    plt.text(x[i], v+0.5, str(v), ha='center', color='red')
plt.show()
```

Explicación:

- `plt.text()` → agrega valor encima de la barra
- `ha='center'` → centra horizontalmente
- `color` → color del texto

Ejercicio 47: Gráfico 3D de superficie

Enunciado:
Crea un gráfico 3D de superficie usando malla de puntos `X` y `Y`.

Solución:

```
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

X = np.linspace(-5,5,20)
Y = np.linspace(-5,5,20)
X, Y = np.meshgrid(X, Y)
Z = np.sin(np.sqrt(X**2 + Y**2))

ax.plot_surface(X, Y, Z, cmap='viridis')
plt.show()
```

Explicación:

- `np.meshgrid()` → crea malla de coordenadas X,Y
- `ax.plot_surface()` → dibuja superficie 3D
- `cmap='viridis'` → mapa de colores

Ejercicio 48: Gráfico combinado con secundario

Enunciado:
Crea gráfico de barras y línea compartiendo eje X, pero línea con eje Y derecho.

Solución:

```
fig, ax1 = plt.subplots()

ax2 = ax1.twinx()
ax1.bar(x, y1, color='lightblue', label='Barras')
ax2.plot(x, y2, color='red', marker='o', label='Línea')

ax1.legend(loc='upper left')
ax2.legend(loc='upper right')
plt.show()
```

Explicación:

- `twinx()` → eje Y secundario
- Diferentes leyendas para cada eje

Ejercicio 49: Gráfico de dispersión con gradiente

Enunciado:
Dibuja scatter con color proporcional a los valores de Y.

Solución:

```
plt.scatter(x, y2, c=y2, cmap='plasma', s=100)
plt.colorbar()
plt.show()
```

Explicación:

- `c=y2` → color según valor
- `cmap='plasma'` → mapa de colores
- `plt.colorbar()` → muestra escala de colores

Ejercicio 50: Animación básica

Enunciado:
Crea animación de un punto moviéndose en línea.

Solución:

```
import matplotlib.animation as animation

fig, ax = plt.subplots()
line, = ax.plot([], [], 'ro')
ax.set_xlim(0,5)
ax.set_ylim(0,25)

def animate(i):
    line.set_data(i, i**2)
    return line,

ani = animation.FuncAnimation(fig, animate, frames=6, interval=500, blit=True)
plt.show()
```

Explicación:

- `FuncAnimation()` → crea animación
- `animate(i)` → función que actualiza datos
- `frames` → número de pasos
- `interval` → tiempo entre frames
- `blit=True` → optimiza la animación

MANUAL EXPLICATIVO DE EJERCICIOS DE IMPRESIÓN

BLOQUE I: Impresión de Texto Simple usando `os.system` (Ejercicios 1-10)

Esta sección utiliza la creación de un archivo temporal de texto (`temp_print.txt`) y el comando del sistema operativo `os.system("notepad /p...")` para enviar el archivo a la impresora predeterminada.

Ejercicio	Enunciado y Solución	Comandos Clave y Controles	Explicación Minuciosa
1	**Imprimir "Hola Mundo"**	`import os`; `open("temp_print.txt", "w")`; `f.write("Hola Mundo")`; `os.system("notepad /p temp_print.txt")`.	El programa primero **crea un archivo de texto** (`temp_print.txt`) con modo de escritura (`"w"`). Escribe el mensaje "Hola Mundo" en este archivo y luego lo **envía a imprimir usando el Bloc de Notas** (`notepad /p`) a través del comando `os.system`.
2	**Imprimir varias líneas**	`\n` (salto de línea).	Se escribe el mensaje en el archivo temporal, utilizando el carácter especial **`\n` para insertar saltos de línea**, permitiendo que el texto se imprima en varias líneas.
3	**Imprimir números del 1 al 10**	`for i in range(1, 11)`; `f.write(f"{i}\n")`.	Se utiliza un bucle `for` junto con `range(1, 11)` para **recorrer los números del 1 al 10**. Cada número (`i`) se escribe en el archivo con un salto de línea (`\n`).
4	**Imprimir tabla de multiplicar**	`5*i` (calcula producto); `f"{5*i}"` (interpolación de cadena).	Se genera la tabla del 5 mediante un bucle. Se usa la expresión `5*i` para calcular el producto y la **interpolación de cadena** (`f"..."`) para formatear la salida.
5	**Imprimir lista de productos**	`zip(productos, precios)`.	Se definen dos listas (productos y precios). La función **`zip()` se usa para emparejar los elementos** correspondientes de ambas listas (`p` y `pr`). El texto se formatea (`f"{p}: ${pr}"`) antes de ser escrito.
6	**Imprimir con fecha y hora**	`import datetime`; `datetime.datetime.now()`.	Se importa el módulo `datetime`. El comando `datetime.datetime.now()` **obtiene la fecha y hora actuales**. Se imprime el mensaje junto con la fecha usando interpolación de cadenas.
7	**Imprimir contenido**	`f.read()`.	Primero, se abre y se lee todo el contenido de un archivo de origen

	de un archivo		(`archivo_origen.txt`) usando **`f.read()`**. Luego, ese contenido se escribe en el archivo temporal de impresión y se envía a imprimir.
8	**Imprimir lista de números pares**	`range(2, 21, 2)`.	El comando **`range(2, 21, 2)` genera números pares** comenzando en 2, hasta antes de 21, con pasos de 2. Cada número par se escribe en una línea separada.
9	**Imprimir mensaje con formato**	`f.write("**** Hola Mundo ****\n")`; Múltiples `f.write()`.	Se utiliza la escritura de cadenas con caracteres de decoración (`****`) para simular un formato de título centrado y líneas separadoras. **Cada `f.write()` imprime una línea**.
10	**Imprimir lista ordenada**	`nombres.sort()`.	Se define una lista de nombres. El método **`nombres.sort()` ordena esta lista alfabéticamente**. Luego, cada nombre ordenado se imprime en una línea.

BLOQUE II: Impresión de Archivos usando `win32api.ShellExecute` (Ejercicios 11-20)

Esta sección utiliza las librerías `win32print` y `win32api` para enviar archivos (creados temporalmente o existentes) directamente a la cola de impresión.

Ejercicio	Enunciado y Solución	Comandos Clave y Controles	Explicación Minuciosa
11	**Imprimir "Hola Mundo" usando win32print**	`import win32print`, `import win32api`; `win32print.GetDefaultPrinter()`; `win32api.ShellExecute(0, "print", file_name, f'/d:"{printer_name}"', ".", 0)`.	Se obtiene el nombre de la impresora predeterminada usando `GetDefaultPrinter()`. Se crea el archivo `temp_print.txt` con el mensaje. Luego, **`ShellExecute` se utiliza con la acción `"print"`** para enviar el archivo al dispositivo especificado (`/d:"{printer_name}"`).
12	**Imprimir varias líneas**	`\n` (salto de línea); `ShellExecute` (para imprimir).	Similar al E2, se usan los caracteres **`\n` para crear saltos de línea** en el archivo temporal, el cual es enviado a imprimir usando `ShellExecute`.
13	**Imprimir números del 1**	`for i in range(1, 11)`.	Se usa el bucle `for` y `range(1, 11)` para

	al 10		**generar los números del 1 al 10**, escribiéndolos en el archivo temporal para su posterior impresión.
14	**Imprimir tabla de multiplicar**	Bucle `for` y cálculo `7*i`.	Genera la tabla del 7. Cada iteración del bucle **escribe una línea con el cálculo de la tabla**.
15	**Imprimir lista de productos con precio**	`zip(productos, precios)`.	Utiliza **`zip()` para emparejar productos y precios** y formatea el texto antes de escribirlo en el archivo temporal para impresión.
16	**Imprimir fecha y hora**	`datetime.datetime.now()`.	Utiliza `datetime.now()` para **obtener la marca de tiempo actual** e incluirla en el mensaje impreso.
17	**Imprimir lista de números pares**	`range(2, 21, 2)`.	El rango `(2, 21, 2)` **genera la secuencia de números pares** del 2 al 20.
18	**Imprimir lista ordenada de nombres**	`nombres.sort()`.	El método **`sort()` ordena la lista de nombres alfabéticamente** antes de que sean escritos línea por línea en el archivo de impresión.
19	**Imprimir tabla de multiplicar personalizada**	`num = int(input("Ingrese un número: "))`; `input()` (recibe número del usuario).	Este ejercicio **recibe un número del usuario** utilizando `input()`. Luego, genera la tabla de multiplicar de ese número (`num*i`) y la imprime.
20	**Imprimir lista de productos ordenada por precio**	`sorted(zip(precios, productos))`; `zip(precios, productos)` (empareja).	Se usa `zip` para emparejar la lista de precios y productos, y luego **`sorted()` ordena la lista basándose en el primer elemento emparejado (el precio)**. El resultado ordenado se imprime.

BLOQUE III: Impresión Directa y Formato usando `win32ui` (Ejercicios 21-30)

Esta sección utiliza el contexto de impresión de Windows (`Device Context` o DC) para dibujar texto directamente en la página de la impresora, sin archivos temporales.

Ejercicio	Enunciado y	Comandos Clave y Controles	Explicación Minuciosa

		Solución	
21	**Imprimir sin crear archivo temporal (texto simple)**	**`win32ui.CreateDC()`**; **`printer_dc.StartDoc(...)`**; **`printer_dc.TextOut(x, y, texto)`**; `EndDoc()`, `DeleteDC()`.	Este flujo abre la impresora (`OpenPrinter`), **crea un contexto de impresión** (`CreateDC`), inicia el documento (`StartDoc`) y la página (`StartPage`). **`TextOut(100, 100, "Hola Mundo")` imprime el texto en las coordenadas (100, 100)**. `DeleteDC()` libera los recursos.
22	**Imprimir varias líneas sin archivo**	`y = 100`; `y += 50` (separa líneas verticalmente).	Se utiliza un bucle para iterar sobre la lista de líneas. La **posición vertical `y` se inicializa y se incrementa en 50 unidades** (`y += 50`) en cada iteración para separar las líneas verticalmente.
23	**Imprimir lista numerada**	`f"{i}. Elemento {i}"`; Incremento de `y`.	Se genera una **lista numerada** (`1. Elemento 1, 2. Elemento 2,` etc.) utilizando interpolación de cadenas dentro de `TextOut`. El incremento de `y` asegura la separación visual.
24	**Imprimir tabla de multiplicar**	`TextOut(100, y, f"3 x {i} = {3*i}")`.	Imprime la tabla del 3. **Cada línea de la tabla se imprime con una separación vertical** de 50 unidades.
25	**Imprimir lista de productos con precios**	`zip` (empareja); `y += 50` (separación vertical).	Se usa `zip` para iterar sobre productos y precios, y `TextOut` imprime el formato deseado en coordenadas que aseguran que **cada elemento esté en una línea diferente**.
26	**Imprimir con negrita y estilo**	**`win32ui.CreateFont({...})`**; **`"weight": 700`** (negrita); **`printer_dc.SelectObject(font)`**.	Se define una fuente (`Arial`, tamaño 50) usando `CreateFont()`. El control `"weight": 700` especifica el estilo **negrita**. `SelectObject(font)` aplica esta fuente al contexto de impresión antes de llamar a `TextOut`.
27	**Imprimir fecha y hora**	`datetime.datetime.now()`; `TextOut` (impresión directa).	Obtiene la fecha y hora usando `datetime.now()` y la imprime directamente usando **`TextOut` en**

			coordenadas específicas.
28	**Imprimir lista con formato**	**`{p:<10}`** (alinea izquierda, 10 caracteres); **`${pr:>5}`** (alinea derecha, 5 caracteres).	Se usa el formato de cadenas F-string para alinear los datos en columnas: `:<10` alinea a la izquierda en 10 espacios, y `:>5` alinea a la derecha en 5 espacios, simulando una tabla formateada.
29	**Imprimir mensajes condicionales**	**`i % 2 == 0`** (operador módulo para paridad); Operador `if` (condicional).	Se utiliza el **operador módulo (%) para determinar la paridad** de un número. Una expresión condicional `if/else` determina si el número es "Par" o "Impar", y el resultado se imprime.
30	**Imprimir tabla completa**	**Anidamiento de bucles** (bucle externo para la tabla, interno para las multiplicaciones); **`{i*j:<4}`** (formato para alinear resultados).	Utiliza bucles anidados: el bucle externo recorre las tablas (1 a 5), y el interno recorre las multiplicaciones (1 a 10). Los resultados se concatenan en una `linea` y se usa el formato `:<4` para alinear los resultados en columnas.

BLOQUE IV: Impresión de Gráficos (Matplotlib) e Imágenes (Ejercicios 31-40)

Esta sección se centra en la generación de visualizaciones usando la librería `matplotlib.pyplot` (`plt`), guardando el resultado como una imagen (`.png`), y luego imprimiéndola usando `ShellExecute`.

Ejercicio	Enunciado y Solución	Comandos Clave y Controles	Explicación Minuciosa
31	**Imprimir gráfico simple como imagen**	**`import matplotlib.pyplot as plt`**; **`plt.bar(x, y)`** (crea barras); **`plt.savefig("grafico.png")`** (guarda imagen); `ShellExecute(..., "print", "grafico.png", ...)`.	Se define `x` e `y`. **`plt.bar()` crea el gráfico de barras**. Crucialmente, `plt.savefig()` guarda el gráfico como un archivo de imagen (`grafico.png`), el cual es enviado a imprimir usando `ShellExecute`.
32	**Imprimir gráfico de línea**	`plt.plot(x, y, marker='o')`; `marker='o'` (puntos en coordenadas).	Se utiliza **`plt.plot()` para generar un gráfico de línea**. El control `marker='o'` asegura que se dibujen puntos en cada coordenada.
33	**Imprimir gráfico de pastel**	`plt.pie(sizes, labels=labels, autopct='%1.1f%%');`	**`plt.pie()` genera el gráfico de pastel**. El formato `autopct` se usa para

		`autopct='%1.1f%%'` (muestra porcentaje).	**mostrar el porcentaje** de cada porción.
34	**Imprimir scatter plot**	`plt.scatter(x, y, color='red');` `plt.scatter()` (dibuja puntos); `color='red'`.	**`plt.scatter()` dibuja un gráfico de dispersión** (puntos). `color='red'` asigna un color específico a los puntos.
35	**Imprimir gráfico con título y etiquetas**	`plt.title("Mi Gráfico");` `plt.xlabel("Eje X");` `plt.ylabel("Eje Y")`.	Se utilizan los comandos **`title()`, `xlabel()` y `ylabel()` para agregar un título y etiquetas** a los ejes del gráfico.
36	**Imprimir gráfico con colores personalizados**	`color=colors`.	Se define una lista de colores y se pasa como argumento `color=` a `plt.bar()`, lo que **asigna diferentes colores a cada barra**.
37	**Imprimir gráfico combinado**	Combinación de `plt.bar()` y `plt.plot()`.	Este ejercicio **combina dos tipos de gráficos** (`plt.bar` para las barras y `plt.plot` para la línea) en la misma figura antes de guardarla para imprimir.
38	**Imprimir gráfico 3D**	**`from mpl_toolkits.mplot3d import Axes3D`**; **`projection='3d'`**; **`ax.bar3d(...)`**.	Requiere importar `Axes3D`. Se crea la figura con la propiedad **`projection='3d'`** para activar la visualización tridimensional. **`ax.bar3d()` dibuja las barras tridimensionales**.
39	**Imprimir imagen desde archivo**	`ShellExecute(0, "print", "mi_imagen.jpg", ...)`.	Simplemente **envía un archivo de imagen existente** (`mi_imagen.jpg`) directamente a la impresora usando `ShellExecute`.
40	**Imprimir varias imágenes secuenciales**	Bucle `for img in imagenes:`.	Se define una lista de archivos de imágenes. Se utiliza un **bucle `for` para recorrer la lista** y ejecutar `ShellExecute()` para imprimir cada archivo de imagen secuencialmente.

BLOQUE V: Reportes y Estructuras Avanzadas (Ejercicios 41-50)

Esta sección combina la impresión directa (`win32ui`) con estructuras de documentos complejas (múltiples páginas, tablas, líneas de separación) y la inclusión de imágenes.

Ejercicio	Enunciado y Solución	Comandos Clave y Controles	Explicación Minuciosa

41	**Imprimir reporte con varias líneas y título**	`TextOut(200, 50, "...")` (Título en coordenadas); Bucle `for` (impresión de datos).	Se utiliza `TextOut` para imprimir un título centrado en coordenadas específicas (200, 50). Luego, se **itera sobre una lista de datos**, imprimiendo cada línea con un incremento en la posición vertical `y`.
42	**Imprimir varias páginas automáticamente**	Bucle `for i in range(1, 4):`; **`printer_dc.StartPage()`**; **`printer_dc.EndPage()`**.	Se usa un bucle para crear 3 páginas. **Cada llamada a `StartPage()` inicia una nueva página** y `EndPage()` la finaliza, permitiendo imprimir contenido diferente en cada una.
43	**Imprimir tabla simple**	`f"{fila:<5} {fila:<5} {fila:<5}"` (formato de columnas); `:<5` (alineación a la izquierda con 5 espacios).	Se define una estructura de datos (lista de listas). Se utiliza el **formato de cadena `:<5` para alinear cada columna a la izquierda** con un ancho de 5 caracteres, construyendo la fila de texto.
44	**Imprimir tabla con encabezados**	`printer_dc.TextOut(100, 50, f"{encabezado:<5} ...")`.	Se imprime el encabezado de las columnas utilizando **`TextOut` en una posición superior (y=50)**. Posteriormente, se imprimen las filas de datos debajo.
45	**Imprimir tabla con línea de separación**	**`printer_dc.Line((100, y, 400, y))`** (dibuja línea horizontal).	Después de imprimir una fila y ajustar la posición `y`, el comando **`Line((x1, y1, x2, y2))` dibuja una línea horizontal** entre las coordenadas especificadas (100, y) y (400, y), separando visualmente la fila.
46	**Imprimir imagen y texto juntos**	**`from PIL import Image`**; `image.save("logo_temp.bmp")`; Impresión de la imagen vía `ShellExecute`; Impresión del texto vía `TextOut`.	Requiere la librería PIL para manejar imágenes. La imagen se convierte a formato BMP (necesario para `ShellExecute` en Windows) y se imprime primero. El texto se imprime luego usando `TextOut`, aunque la solución del código lo hace en páginas separadas.
47	**Imprimir varias imágenes en una página**	Bucle `for img in imagenes:`; `y += 300` (ajuste de posición).	Se itera sobre una lista de imágenes. Aunque la explicación menciona ajustar

			la posición `y`, el código fuente utiliza `ShellExecute` para imprimir cada imagen, lo cual **normalmente inicia trabajos de impresión separados**.
48	**Imprimir mensaje condicional en una tabla**	`tipo = "Par" if i%2==0 else "Impar"`; `i%2==0` (determinación de paridad).	Similar al E29, pero en formato de tabla simple. Utiliza el operador `%` para **determinar si el número es par o impar** y lo imprime en la columna de la derecha.
49	**Imprimir múltiples páginas con tablas**	Bucle `for pagina in range(2):`; `printer_dc.StartPage()`.	Se utiliza un bucle externo (`for pagina in range(2)`) para generar dos páginas. Dentro de este bucle, **se inicia una nueva página con `StartPage()`** y se imprime el contenido de la tabla en cada una.
50	**Reporte completo con texto, tabla e imagen**	Combina `TextOut` (título y tabla) y `ShellExecute` (imagen).	Este ejercicio **combina múltiples técnicas**: `TextOut` imprime el título y la tabla de datos en la página, y una vez que el contexto de impresión de texto termina (`EndDoc`), **`ShellExecute` se usa por separado para enviar la imagen** (`grafico.png`) a imprimir.

Ejercicio 1: Imprimir "Hola Mundo"

Enunciado:
Crea un programa que imprima el mensaje `"Hola Mundo"` en la impresora predeterminada.

Solución:

```
import os

with open("temp_print.txt", "w") as f:
    f.write("Hola Mundo")

os.system("notepad /p temp_print.txt")
```

Explicación:

- `open("temp_print.txt", "w")` → crea un archivo de texto
- `f.write("Hola Mundo")` → escribe el mensaje
- `os.system("notepad /p temp_print.txt")` → envía a imprimir usando Bloc de notas

Ejercicio 2: Imprimir varias líneas

Enunciado:
Imprime un mensaje con varias líneas de texto.

Solución:

```
with open("temp_print.txt", "w") as f:
    f.write("Hola Mundo\nEsta es una prueba\nPython imprime")

os.system("notepad /p temp_print.txt")
```

Explicación:

- `\n` → salto de línea

Ejercicio 3: Imprimir números del 1 al 10

Enunciado:
Imprime los números del 1 al 10 en la impresora.

Solución:

```
with open("temp_print.txt", "w") as f:
    for i in range(1, 11):
        f.write(f"{i}\n")

os.system("notepad /p temp_print.txt")
```

Explicación:

- `for i in range(1,11)` → recorre del 1 al 10
- `f.write(f"{i}\n")` → imprime cada número con salto de línea

Ejercicio 4: Imprimir tabla de multiplicar

Enunciado:
Imprime la tabla del 5 en la impresora.

Solución:

```
with open("temp_print.txt", "w") as f:
    for i in range(1, 11):
        f.write(f"5 x {i} = {5*i}\n")

os.system("notepad /p temp_print.txt")
```

Explicación:

- `5*i` → calcula producto
- `f"{5*i}"` → interpolación de cadena

Ejercicio 5: Imprimir lista de productos

Enunciado:
Imprime una lista de productos con su precio.

Solución:

```
productos = ["Leche", "Pan", "Queso"]
precios = [1.5, 0.8, 2.3]

with open("temp_print.txt", "w") as f:
    for p, pr in zip(productos, precios):
        f.write(f"{p}: ${pr}\n")

os.system("notepad /p temp_print.txt")
```

Explicación:

- `zip(productos, precios)` → empareja elementos
- `f"{p}: ${pr}"` → formatea texto

Ejercicio 6: Imprimir con fecha y hora

Enunciado:
Imprime un mensaje con la fecha y hora actuales.

Solución:

```
import datetime

fecha = datetime.datetime.now()

with open("temp_print.txt", "w") as f:
    f.write(f"Hola Mundo\nFecha y hora: {fecha}\n")

os.system("notepad /p temp_print.txt")
```

Explicación:

- `datetime.datetime.now()` → obtiene fecha y hora
- Se imprime usando interpolación de cadenas

Ejercicio 7: Imprimir contenido de un archivo

Enunciado:
Lee un archivo de texto y envíalo a la impresora.

Solución:

```
with open("archivo_origen.txt", "r") as f:
    contenido = f.read()

with open("temp_print.txt", "w") as f:
    f.write(contenido)

os.system("notepad /p temp_print.txt")
```

Explicación:

- `f.read()` → lee todo el contenido del archivo
- Se guarda en archivo temporal y se imprime

Ejercicio 8: Imprimir lista de números pares

Enunciado:
Imprime los números pares del 1 al 20.

Solución:

```
with open("temp_print.txt", "w") as f:
    for i in range(2, 21, 2):
        f.write(f"{i}\n")

os.system("notepad /p temp_print.txt")
```

Explicación:

- `range(2,21,2)` → genera números pares de 2 a 20

Ejercicio 9: Imprimir mensaje con formato

Enunciado:
Imprime un mensaje con título centrado y líneas separadoras.

Solución:

```
with open("temp_print.txt", "w") as f:
    f.write("**** Hola Mundo ****\n")
    f.write("Mensaje de prueba de impresión\n")
    f.write("**********************\n")

os.system("notepad /p temp_print.txt")
```

Explicación:

- `****` → decoraciones
- Cada `f.write()` imprime una línea

Ejercicio 10: Imprimir lista ordenada

Enunciado:
Imprime una lista de nombres ordenados alfabéticamente.

Solución:

```
nombres = ["Ana", "Luis", "Eva"]
nombres.sort()

with open("temp_print.txt", "w") as f:
    for nombre in nombres:
        f.write(f"{nombre}\n")

os.system("notepad /p temp_print.txt")
```

Explicación:

- `nombres.sort()` → ordena alfabéticamente
- Se imprime cada nombre en una línea

Ejercicio 11: Imprimir "Hola Mundo" usando win32print

Enunciado:
Imprime el mensaje `"Hola Mundo"` usando la impresora predeterminada con `win32print`.

Solución:

```
import win32print
import win32api

printer_name = win32print.GetDefaultPrinter()
file_name = "temp_print.txt"

with open(file_name, "w") as f:
    f.write("Hola Mundo")

win32api.ShellExecute(0, "print", file_name, f'/d:"{printer_name}"', ".", 0)
```

Explicación:

- `GetDefaultPrinter()` → obtiene impresora predeterminada
- `ShellExecute(..., "print", ...)` → envía archivo a imprimir

Ejercicio 12: Imprimir varias líneas

Enunciado:
Envía a imprimir un mensaje con varias líneas usando win32print.

Solución:

```
with open("temp_print.txt", "w") as f:
    f.write("Hola Mundo\nEsta es una prueba\nPython imprime")

win32api.ShellExecute(0, "print", "temp_print.txt", None, ".", 0)
```

Explicación:

- `\n` → salto de línea
- Se usa `ShellExecute` para imprimir

Ejercicio 13: Imprimir números del 1 al 10

Enunciado:
Imprime los números del 1 al 10 usando win32api.

Solución:

```
with open("temp_print.txt", "w") as f:
    for i in range(1,11):
        f.write(f"{i}\n")

win32api.ShellExecute(0, "print", "temp_print.txt", None, ".", 0)
```

Explicación:

- `for i in range(1,11)` → genera números del 1 al 10

Ejercicio 14: Imprimir tabla de multiplicar

Enunciado:
Imprime la tabla del 7 en la impresora.

Solución:

```
with open("temp_print.txt", "w") as f:
    for i in range(1, 11):
        f.write(f"7 x {i} = {7*i}\n")

win32api.ShellExecute(0, "print", "temp_print.txt", None, ".", 0)
```

Explicación:

- Cada línea imprime un cálculo de la tabla

Ejercicio 15: Imprimir lista de productos con precio

Enunciado:

Imprime productos con precios usando win32api.

Solución:

```
productos = ["Leche", "Pan", "Queso"]
precios = [1.5, 0.8, 2.3]

with open("temp_print.txt", "w") as f:
    for p, pr in zip(productos, precios):
        f.write(f"{p}: ${pr}\n")

win32api.ShellExecute(0, "print", "temp_print.txt", None, ".", 0)
```

Explicación:

- `zip(productos, precios)` → empareja elementos

Ejercicio 16: Imprimir fecha y hora

Enunciado:
Imprime un mensaje con la fecha y hora actuales.

Solución:

```
import datetime

fecha = datetime.datetime.now()

with open("temp_print.txt", "w") as f:
    f.write(f"Hola Mundo\nFecha y hora: {fecha}\n")

win32api.ShellExecute(0, "print", "temp_print.txt", None, ".", 0)
```

Explicación:

- `datetime.now()` → obtiene fecha y hora

Ejercicio 17: Imprimir lista de números pares

Enunciado:
Imprime los números pares del 2 al 20.

Solución:

```
with open("temp_print.txt", "w") as f:
    for i in range(2,21,2):
        f.write(f"{i}\n")

win32api.ShellExecute(0, "print", "temp_print.txt", None, ".", 0)
```

Explicación:

- `range(2,21,2)` → genera números pares

Ejercicio 18: Imprimir lista ordenada de nombres

Enunciado:
Imprime nombres ordenados alfabéticamente.

Solución:

```
nombres = ["Ana", "Luis", "Eva"]
nombres.sort()

with open("temp_print.txt", "w") as f:
    for nombre in nombres:
        f.write(f"{nombre}\n")

win32api.ShellExecute(0, "print", "temp_print.txt", None, ".", 0)
```

Explicación:

- `sort()` → ordena alfabéticamente

Ejercicio 19: Imprimir tabla de multiplicar personalizada

Enunciado:
Imprime la tabla de cualquier número ingresado por el usuario.

Solución:

```
num = int(input("Ingrese un número: "))

with open("temp_print.txt", "w") as f:
    for i in range(1, 11):
        f.write(f"{num} x {i} = {num*i}\n")

win32api.ShellExecute(0, "print", "temp_print.txt", None, ".", 0)
```

Explicación:

- `input()` → recibe número del usuario
- Se genera la tabla y se imprime

Ejercicio 20: Imprimir lista de productos ordenada por precio

Enunciado:
Imprime productos ordenados de menor a mayor precio.

Solución:

```
productos = ["Leche", "Pan", "Queso"]
precios = [1.5, 0.8, 2.3]

productos_ordenados = [p for _, p in sorted(zip(precios, productos))]

with open("temp_print.txt", "w") as f:
    for p in productos_ordenados:
        f.write(f"{p}\n")

win32api.ShellExecute(0, "print", "temp_print.txt", None, ".", 0)
```

Explicación:

- `zip(precios, productos)` → empareja precios y productos
- `sorted()` → ordena por precio

Ejercicio 21: Imprimir sin crear archivo temporal (texto simple)

Enunciado:
Envía directamente el texto `"Hola Mundo"` a la impresora sin crear un archivo.

Solución:

```
import win32print
import win32ui

printer_name = win32print.GetDefaultPrinter()
hprinter = win32print.OpenPrinter(printer_name)
printer_dc = win32ui.CreateDC()
printer_dc.CreatePrinterDC(printer_name)
printer_dc.StartDoc("Documento Python")
printer_dc.StartPage()
printer_dc.TextOut(100, 100, "Hola Mundo")
printer_dc.EndPage()
printer_dc.EndDoc()
printer_dc.DeleteDC()
```

Explicación:

- `OpenPrinter()` → abre impresora
- `CreateDC()` → crea contexto de impresión
- `StartDoc()` → inicia documento
- `StartPage()` / `EndPage()` → inicia y termina página
- `TextOut(x, y, texto)` → imprime texto en coordenadas
- `EndDoc()` → finaliza documento
- `DeleteDC()` → libera recursos

Ejercicio 22: Imprimir varias líneas sin archivo

Enunciado:
Imprime 5 líneas de texto usando coordenadas diferentes.

Solución:

```
lineas = ["Linea 1", "Linea 2", "Linea 3", "Linea 4", "Linea 5"]

printer_name = win32print.GetDefaultPrinter()
printer_dc = win32ui.CreateDC()
printer_dc.CreatePrinterDC(printer_name)
printer_dc.StartDoc("Varias Líneas")
printer_dc.StartPage()

y = 100
for linea in lineas:
    printer_dc.TextOut(100, y, linea)
    y += 50  # separa líneas verticalmente

printer_dc.EndPage()
printer_dc.EndDoc()
printer_dc.DeleteDC()
```

Explicación:

- Se ajusta la posición `y` para cada línea
- `TextOut` dibuja el texto en la impresora

Ejercicio 23: Imprimir lista numerada

Enunciado:
Imprime los números del 1 al 10 en formato de lista.

Solución:

```
printer_name = win32print.GetDefaultPrinter()
printer_dc = win32ui.CreateDC()
printer_dc.CreatePrinterDC(printer_name)
printer_dc.StartDoc("Lista Numerada")
printer_dc.StartPage()

y = 100
for i in range(1, 11):
    printer_dc.TextOut(100, y, f"{i}. Elemento {i}")
    y += 50

printer_dc.EndPage()
printer_dc.EndDoc()
printer_dc.DeleteDC()
```

Explicación:

- Lista numerada usando `f"{i}. Elemento {i}"`
- Incremento de `y` separa las líneas

Ejercicio 24: Imprimir tabla de multiplicar

Enunciado:
Imprime la tabla del 3 usando coordenadas automáticas.

Solución:

```
printer_name = win32print.GetDefaultPrinter()
printer_dc = win32ui.CreateDC()
printer_dc.CreatePrinterDC(printer_name)
printer_dc.StartDoc("Tabla del 3")
printer_dc.StartPage()

y = 100
for i in range(1, 11):
    printer_dc.TextOut(100, y, f"3 x {i} = {3*i}")
    y += 50

printer_dc.EndPage()
printer_dc.EndDoc()
printer_dc.DeleteDC()
```

Explicación:

- Cada línea de la tabla se imprime con separación vertical

Ejercicio 25: Imprimir lista de productos con precios

Enunciado:
Imprime productos y precios sin archivo temporal.

Solución:

```
productos = ["Leche", "Pan", "Queso"]
precios = [1.5, 0.8, 2.3]

printer_name = win32print.GetDefaultPrinter()
printer_dc = win32ui.CreateDC()
printer_dc.CreatePrinterDC(printer_name)
printer_dc.StartDoc("Productos")
printer_dc.StartPage()

y = 100
for p, pr in zip(productos, precios):
    printer_dc.TextOut(100, y, f"{p}: ${pr}")
    y += 50

printer_dc.EndPage()
printer_dc.EndDoc()
printer_dc.DeleteDC()
```

Explicación:

- `zip` empareja productos con precios
- Cada línea se imprime con separación vertical

Ejercicio 26: Imprimir con negrita y estilo

Enunciado:
Imprime texto en negrita y fuente Arial.

Solución:

```
printer_name = win32print.GetDefaultPrinter()
printer_dc = win32ui.CreateDC()
printer_dc.CreatePrinterDC(printer_name)
printer_dc.StartDoc("Texto Estilizado")
printer_dc.StartPage()

font = win32ui.CreateFont({
    "name": "Arial",
    "height": 50,
    "weight": 700  # negrita
})
printer_dc.SelectObject(font)
printer_dc.TextOut(100, 100, "Hola Mundo Estilizado")

printer_dc.EndPage()
printer_dc.EndDoc()
printer_dc.DeleteDC()
```

Explicación:

- `CreateFont()` → define fuente, tamaño y estilo
- `weight=700` → negrita
- `SelectObject(font)` → aplica fuente al contexto de impresión

Ejercicio 27: Imprimir fecha y hora

Enunciado:
Imprime un mensaje con la fecha y hora actuales.

Solución:

```
import datetime
fecha = datetime.datetime.now()

printer_name = win32print.GetDefaultPrinter()
printer_dc = win32ui.CreateDC()
printer_dc.CreatePrinterDC(printer_name)
printer_dc.StartDoc("Fecha y Hora")
printer_dc.StartPage()

printer_dc.TextOut(100, 100, f"Hola Mundo - {fecha}")
printer_dc.EndPage()
printer_dc.EndDoc()
printer_dc.DeleteDC()
```

Explicación:

- `datetime.datetime.now()` → obtiene fecha y hora
- Se imprime directamente usando `TextOut`

Ejercicio 28: Imprimir lista con formato

Enunciado:
Imprime productos en columnas: producto y precio alineados.

Solución:

```
productos = ["Leche", "Pan", "Queso"]
precios = [1.5, 0.8, 2.3]

printer_name = win32print.GetDefaultPrinter()
printer_dc = win32ui.CreateDC()
printer_dc.CreatePrinterDC(printer_name)
printer_dc.StartDoc("Tabla Formateada")
printer_dc.StartPage()

y = 100
for p, pr in zip(productos, precios):
    printer_dc.TextOut(100, y, f"{p:<10} ${pr:>5}")
    y += 50

printer_dc.EndPage()
printer_dc.EndDoc()
printer_dc.DeleteDC()
```

Explicación:

- `:<10` → alinea a la izquierda con 10 caracteres
- `:>5` → alinea a la derecha con 5 caracteres

Ejercicio 29: Imprimir mensajes condicionales

Enunciado:

Imprime si los números del 1 al 10 son pares o impares.

Solución:

```
printer_name = win32print.GetDefaultPrinter()
printer_dc = win32ui.CreateDC()
printer_dc.CreatePrinterDC(printer_name)
printer_dc.StartDoc("Pares e Impares")
printer_dc.StartPage()

y = 100
for i in range(1, 11):
    tipo = "Par" if i % 2 == 0 else "Impar"
    printer_dc.TextOut(100, y, f"{i} es {tipo}")
    y += 50

printer_dc.EndPage()
printer_dc.EndDoc()
printer_dc.DeleteDC()
```

Explicación:

- Operador `%` → módulo para determinar paridad
- `if` → condicional para definir tipo

Ejercicio 30: Imprimir tabla completa

Enunciado:
Imprime tablas de multiplicar del 1 al 5 en columnas.

Solución:

```
printer_name = win32print.GetDefaultPrinter()
printer_dc = win32ui.CreateDC()
printer_dc.CreatePrinterDC(printer_name)
printer_dc.StartDoc("Tablas 1-5")
printer_dc.StartPage()

y = 100
for i in range(1, 6):
    linea = ""
    for j in range(1, 11):
        linea += f"{i}x{j}={i*j:<4} "
    printer_dc.TextOut(100, y, linea)
    y += 50

printer_dc.EndPage()
printer_dc.EndDoc()
printer_dc.DeleteDC()
```

Explicación:

- Anidamiento de bucles para columnas
- `:<4` → formato para alinear resultados

Ejercicio 31: Imprimir gráfico simple como imagen

Enunciado:
Crea un gráfico de barras y envíalo a imprimir como imagen.

Solución:

```
import matplotlib.pyplot as plt
import win32api
import win32print

x = [1,2,3,4]
y = [10,20,15,25]

plt.bar(x, y)
plt.savefig("grafico.png")  # guarda gráfico como imagen

printer_name = win32print.GetDefaultPrinter()
win32api.ShellExecute(0, "print", "grafico.png", f'/d:"{printer_name}"', ".", 0)
```

Explicación:

- `plt.bar()` → crea gráfico de barras
- `plt.savefig()` → guarda gráfico en archivo
- `ShellExecute(..., "print", ...)` → envía imagen a impresora

Ejercicio 32: Imprimir gráfico de línea

Enunciado:
Imprime gráfico de línea con varios puntos.

Solución:

```
x = [1,2,3,4,5]
y = [2,4,6,8,10]

plt.plot(x, y, marker='o')
plt.savefig("linea.png")

printer_name = win32print.GetDefaultPrinter()
win32api.ShellExecute(0, "print", "linea.png", f'/d:"{printer_name}"', ".", 0)
```

Explicación:

- `marker='o'` → puntos en cada coordenada
- Se imprime la imagen guardada

Ejercicio 33: Imprimir gráfico de pastel

Enunciado:
Imprime gráfico de pastel mostrando proporciones.

Solución:

```
labels = ["A", "B", "C"]
sizes = [30, 50, 20]

plt.pie(sizes, labels=labels, autopct='%1.1f%%')
plt.savefig("pastel.png")

printer_name = win32print.GetDefaultPrinter()
win32api.ShellExecute(0, "print", "pastel.png", f'/d:"{printer_name}"', ".", 0)
```

Explicación:

- `plt.pie()` → gráfico de pastel
- `autopct='%1.1f%%'` → muestra porcentaje

Ejercicio 34: Imprimir scatter plot

Enunciado:
Imprime gráfico de dispersión (scatter plot).

Solución:

```
x = [1,2,3,4]
y = [4,7,5,8]

plt.scatter(x, y, color='red')
plt.savefig("scatter.png")

win32api.ShellExecute(0, "print", "scatter.png", f'/d:"{printer_name}"', ".", 0)
```

Explicación:

- `plt.scatter()` → dibuja puntos
- `color='red'` → color de los puntos

Ejercicio 35: Imprimir gráfico con título y etiquetas

Enunciado:

Imprime gráfico de barras con título y etiquetas de ejes.

Solución:

```
plt.bar(x, y)
plt.title("Mi Gráfico")
plt.xlabel("Eje X")
plt.ylabel("Eje Y")
plt.savefig("grafico_etiquetas.png")
win32api.ShellExecute(0, "print", "grafico_etiquetas.png",
f'/d:"{printer_name}"', ".", 0)
```

Explicación:

- `title()`, `xlabel()`, `ylabel()` → añaden título y etiquetas

Ejercicio 36: Imprimir gráfico con colores personalizados

Enunciado:

Imprime barras con diferentes colores.

Solución:

```
colors = ['red', 'green', 'blue', 'orange']
plt.bar(x, y, color=colors)
plt.savefig("grafico_colores.png")
win32api.ShellExecute(0, "print", "grafico_colores.png", f'/d:"{printer_name}"', ".", 0)
```

Explicación:

- `color=colors` → asigna colores a cada barra

Ejercicio 37: Imprimir gráfico combinado

Enunciado:

Imprime barras y línea en el mismo gráfico.

Solución:

```
plt.bar(x, y, color='lightblue')
plt.plot(x, y, color='red', marker='o')
plt.savefig("grafico_combinado.png")
win32api.ShellExecute(0, "print", "grafico_combinado.png",
f'/d:"{printer_name}"', ".", 0)
```

Explicación:

- Combinación de `plt.bar()` y `plt.plot()`

Ejercicio 38: Imprimir gráfico 3D

Enunciado:
Imprime gráfico 3D de barras.

Solución:

```
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x = [1,2,3,4]
y = [1,2,3,4]
z = np.zeros(4)
dx = dy = 0.5
dz = [5,10,7,3]

ax.bar3d(x, y, z, dx, dy, dz, color='cyan')
plt.savefig("grafico3d.png")
win32api.ShellExecute(0, "print", "grafico3d.png", f'/d:"{printer_name}"', ".",
0)
```

Explicación:

- `Axes3D` → activa gráfico 3D
- `bar3d()` → barras tridimensionales

Ejercicio 39: Imprimir imagen desde archivo

Enunciado:
Imprime cualquier imagen desde disco.

Solución:

```
win32api.ShellExecute(0, "print", "mi_imagen.jpg", f'/d:"{printer_name}"', ".", 0)
```

Explicación:

- Solo envía archivo de imagen a la impresora

Ejercicio 40: Imprimir varias imágenes secuenciales

Enunciado:

Imprime varias imágenes en secuencia.

Solución:

```
imagenes = ["grafico.png", "linea.png", "pastel.png"]
for img in imagenes:
    win32api.ShellExecute(0, "print", img, f'/d:"{printer_name}"', ".", 0)
```

Explicación:

- Bucle `for` recorre la lista de imágenes
- `ShellExecute()` imprime cada archivo

Ejercicio 41: Imprimir reporte con varias líneas y título

Enunciado:
Imprime un reporte con título centrado y varias líneas de datos.

Solución:

```
printer_name = win32print.GetDefaultPrinter()
printer_dc = win32ui.CreateDC()
printer_dc.CreatePrinterDC(printer_name)
printer_dc.StartDoc("Reporte Avanzado")
printer_dc.StartPage()

printer_dc.TextOut(200, 50, "**** REPORTE DE DATOS ****")
lineas = ["Dato 1: 100", "Dato 2: 200", "Dato 3: 300"]
y = 150
for linea in lineas:
    printer_dc.TextOut(100, y, linea)
    y += 50

printer_dc.EndPage()
printer_dc.EndDoc()
printer_dc.DeleteDC()
```

Explicación:

- `TextOut(x,y,texto)` → imprime en coordenadas específicas
- `StartDoc()` / `EndDoc()` → inicia y termina el documento
- `StartPage()` / `EndPage()` → controla páginas

Ejercicio 42: Imprimir varias páginas automáticamente

Enunciado:
Crea un documento de 3 páginas con texto diferente en cada una.

Solución:

```
printer_name = win32print.GetDefaultPrinter()
printer_dc = win32ui.CreateDC()
printer_dc.CreatePrinterDC(printer_name)
printer_dc.StartDoc("Documento Multipágina")

for i in range(1, 4):
    printer_dc.StartPage()
    printer_dc.TextOut(100, 100, f"Página {i}: Contenido de prueba")
    printer_dc.EndPage()

printer_dc.EndDoc()
printer_dc.DeleteDC()
```

Explicación:

- Cada `StartPage()` → nueva página
- Se imprime contenido distinto en cada página

Ejercicio 43: Imprimir tabla simple

Enunciado:
Imprime una tabla de 3 columnas y 5 filas.

Solución:

```
datos = [["A1","B1","C1"], ["A2","B2","C2"], ["A3","B3","C3"], ["A4","B4","C4"], ["A5","B5","C5"]]


printer_name = win32print.GetDefaultPrinter()
printer_dc = win32ui.CreateDC()
printer_dc.CreatePrinterDC(printer_name)
printer_dc.StartDoc("Tabla Simple")
printer_dc.StartPage()


y = 100
for fila in datos:
    texto = f"{fila[0]:<5} {fila[1]:<5} {fila[2]:<5}"
    printer_dc.TextOut(100, y, texto)
    y += 50


printer_dc.EndPage()
printer_dc.EndDoc()
printer_dc.DeleteDC()
```

Explicación:

- `:<5` → alinea cada columna a la izquierda
- Se construye línea de texto por fila

Ejercicio 44: Imprimir tabla con encabezados

Enunciado:
Agrega encabezado de columnas a la tabla.

Solución:

```
encabezado = ["Col1","Col2","Col3"]

printer_dc.StartPage()

printer_dc.TextOut(100, 50, f"{encabezado[0]:<5} {encabezado[1]:<5} {encabezado[2]:<5}")


y = 100

for fila in datos:

    texto = f"{fila[0]:<5} {fila[1]:<5} {fila[2]:<5}"

    printer_dc.TextOut(100, y, texto)

    y += 50


printer_dc.EndPage()
```

Explicación:

- Primer `TextOut()` → imprime encabezado
- Después se imprimen filas

Ejercicio 45: Imprimir tabla con línea de separación

Enunciado:
Dibuja líneas horizontales entre filas de la tabla.

Solución:

```
y = 100
for fila in datos:
    texto = f"{fila[0]:<5} {fila[1]:<5} {fila[2]:<5}"
    printer_dc.TextOut(100, y, texto)
    y += 40
    printer_dc.Line((100, y, 400, y))  # línea horizontal
    y += 10
```

Explicación:

- `Line((x1,y1,x2,y2))` → dibuja línea entre coordenadas

Ejercicio 46: Imprimir imagen y texto juntos

Enunciado:
Imprime un logo y un texto debajo.

Solución:

```
from PIL import Image


image = Image.open("logo.png")
image.save("logo_temp.bmp")  # Windows necesita BMP para ShellExecute


win32api.ShellExecute(0, "print", "logo_temp.bmp", f'/d:"{printer_name}"', ".",
0)


printer_dc.StartDoc("Imagen y Texto")
printer_dc.StartPage()
printer_dc.TextOut(100, 200, "Este es el texto debajo del logo")
printer_dc.EndPage()
printer_dc.EndDoc()
```

Explicación:

- `PIL.Image.open()` → abre imagen
- Se imprime imagen y texto en páginas separadas

Ejercicio 47: Imprimir varias imágenes en una página

Enunciado:
Coloca varias imágenes en distintas posiciones de la misma página.

Solución:

```
imagenes = ["grafico.png", "linea.png"]
y = 100
for img in imagenes:
    win32api.ShellExecute(0, "print", img, f'/d:"{printer_name}"', ".", 0)
    y += 300
```

Explicación:

- Se ajusta posición `y` para cada imagen
- `ShellExecute()` imprime cada imagen

Ejercicio 48: Imprimir mensaje condicional

Enunciado:
Imprime si los números del 1 al 10 son pares o impares en una tabla.

Solución:

```
y = 100
for i in range(1,11):
    tipo = "Par" if i%2==0 else "Impar"
    printer_dc.TextOut(100, y, f"{i}: {tipo}")
    y += 50
```

Explicación:

- Operador `%` determina paridad
- `if` aplica condición para cada fila

Ejercicio 49: Imprimir múltiples páginas con tablas

Enunciado:
Crea 2 páginas de tablas de datos.

Solución:

```
for pagina in range(2):
    printer_dc.StartPage()
    y = 100
    for fila in datos:
        texto = f"{fila[0]} {fila[1]} {fila[2]}"
        printer_dc.TextOut(100, y, texto)
        y += 50
    printer_dc.EndPage()
```

Explicación:

- Cada `StartPage()` → nueva página
- Repetición de filas por página

Ejercicio 50: Reporte completo con texto, tabla e imagen

Enunciado:
Crea un reporte con título, tabla de datos y una imagen al final.

Solución:

```
printer_dc.StartDoc("Reporte Completo")
printer_dc.StartPage()

printer_dc.TextOut(200, 50, "**** REPORTE COMPLETO ****")

# Tabla
y = 150
for fila in datos:
    texto = f"{fila[0]:<5} {fila[1]:<5} {fila[2]:<5}"
    printer_dc.TextOut(100, y, texto)
    y += 50

printer_dc.EndPage()
printer_dc.EndDoc()

# Imagen
win32api.ShellExecute(0, "print", "grafico.png", f'/d:"{printer_name}"', ".", 0)
printer_dc.DeleteDC()
```

Explicación:

- Se imprime título, tabla y finalmente imagen
- Combina `TextOut` y `ShellExecute`

MANUAL DETALLADO DE EJERCICIOS DE PYTHON CON COLORAMA

Configuración Inicial Común a Todos los Ejercicios

Para utilizar colores en la consola, todos los ejercicios requieren la importación e inicialización de la librería `colorama`.

Comando	Explicación del Control
`from colorama import Fore, Style, init`	Importa las funciones necesarias (`Fore` para colores, `init` para inicialización).
`init(autoreset=True)`	Inicializa la librería. **`autoreset=True`** garantiza que el color se reinicie automáticamente después de cada impresión (`print`).
`Fore.COLOR`	Prefijo utilizado antes del texto para cambiar su color (Ejemplos: `Fore.RED`, `Fore.BLUE`, `Fore.GREEN`, `Fore.CYAN`, `Fore.YELLOW`, `Fore.MAGENTA`).

Ejercicios 1 al 10

Ejercicio	Enunciado y Código	Controles y Comandos Clave
1 – Hola Mundo en color	Mostrar "Hola Mundo" en color rojo. `print(Fore.RED + "Hola Mundo")`.	**`Fore.RED`**: Cambia el color del texto a rojo.
2 – Suma de dos números	Pedir dos números, sumarlos y mostrar el resultado en azul.	**`float(input(...))`**: Convierte la entrada del usuario a un número decimal (flotante). **`Fore.BLUE`**: Imprime el resultado en color azul.
3 – Comparar dos números	Recibir dos números. Mostrar mensaje en verde (A>B), amarillo (B>A) o rojo (iguales).	**`if/elif/else`**: Estructura condicional que ejecuta diferentes bloques de código según la condición. Se usan **`Fore.GREEN`**, **`Fore.YELLOW`** y **`Fore.RED`** según el resultado.
4 – Tabla de multiplicar	Mostrar la tabla de multiplicar de un número introducido por el usuario en color cyan.	**`for i in range(1, 11)`**: Bucle que repite la operación 10 veces, y la variable `i` toma valores del 1 al 10. **`f"{num} x {i} = {num*i}"`**: Uso de *f-string* para formatear la impresión del resultado. **`Fore.CYAN`** aplica el color.
5 – Contador de letras	Contar cuántas veces aparece una letra en un texto. Resultado en magenta.	**`texto.count(letra)`**: Método de cadena (string) que cuenta el número de veces que aparece un subtexto (`letra`) dentro del `texto`. **`Fore.MAGENTA`** aplica el color.
6 – Números pares del 1 al 20	Mostrar todos los números pares del 1 al 20 en verde.	**`i % 2 == 0`**: Operador módulo (%) que devuelve el resto de la división. Si el resto es 0, el número es par. **`str(i)`**: Convierte el número a texto para que la aplicación del color funcione correctamente.
7 – Lista de nombres	Pedir 5 nombres y mostrarlos en azul.	**`for _ in range(5)`**: Repite la petición de nombres 5 veces. El guion bajo (_) se usa como variable cuando no se necesita su valor dentro del bucle.

		`nombres.append(nombre)`: Agrega el nombre introducido al final de la lista `nombres`.
8 – Suma de lista de números	Sumar todos los números de una lista predefinida y mostrar el resultado en amarillo.	**`sum(numeros)`**: Función incorporada que calcula la suma de todos los elementos de la lista `numeros`. **`Fore.YELLOW`** muestra el resultado.
9 – Número primo	Determinar si un número es primo. Resultado en cyan si es primo, rojo si no lo es.	**`int(num**0.5)+1`**: Optimiza el bucle de comprobación de divisores, revisando solo hasta la raíz cuadrada del número. **`break`**: Comando que fuerza la salida inmediata del bucle si se encuentra un divisor (indicando que no es primo).
10 – Invertir texto	Invertir un texto introducido por el usuario y mostrarlo en magenta.	**`texto[::-1]`**: Técnica de *slicing* (rebanado) que invierte la cadena de texto al recorrerla de atrás hacia adelante.

Ejercicios 11 al 20

Ejercicio	Enunciado y Código	Controles y Comandos Clave
11 – Contador de vocales	Contar cuántas vocales hay en un texto y mostrar el resultado en verde.	**`sum(1 for letra in texto if letra in vocales)`**: Uso de una expresión generadora que suma 1 por cada carácter que se encuentra dentro de la cadena `vocales`.
12 – Lista al cuadrado	Mostrar una lista con los números al cuadrado de una lista original en azul.	**`[x**2 for x in numeros]`**: Lista por comprensión que eleva cada elemento (`x`) de la lista `numeros` al cuadrado.
13 – Media de números	Calcular la media (promedio) de una lista de números. Resultado en amarillo.	**`len(numeros)`**: Función que devuelve la cantidad de elementos de la lista, necesaria para dividir la suma y obtener la media.
14 – Palíndromo	Comprobar si un texto es palíndromo (se lee igual al derecho y al revés). Cyan si lo es, rojo si no.	**`.replace(" ", "").lower()`**: Normaliza el texto eliminando espacios y convirtiéndolo a minúsculas, asegurando una comparación precisa de palíndromos.
15 – Números impares y pares en colores	Recorrer números del 1 al 10. Pares en verde, impares en rojo.	El código combina el operador módulo (%) con una estructura `if/else` para aplicar **`Fore.GREEN`** o **`Fore.RED`** a cada número.
16 – Función que saluda	Crear una función que salude a un nombre en color azul.	**`def saludar(nombre):`**: Define una función que toma `nombre` como parámetro. **`saludar(nombre)`**: Llama (ejecuta) la función.
17 – Números primos en lista	Generar todos los números primos del 1 al 50 y mostrarlos en cyan.	**`num**0.5`**: Se usa la raíz cuadrada del número para optimizar el chequeo de primalidad. **`primos.append(num)`**: Añade el número si se determina que es primo.
18 – Diccionario de edades	Mostrar nombres y edades de un diccionario. Verde si >= 18, amarillo si menor.	**`for nombre, edad in edades.items():`**: Bucle que recorre simultáneamente las claves (`nombre`) y los valores (`edad`) del diccionario. **`Fore.GREEN`** o

		`Fore.YELLOW` se eligen condicionalmente.
19 – Suma de números pares en lista	Sumar solo los números pares de una lista. Resultado en azul.	**`sum(x for x in numeros if x % 2 == 0)`**: Usa una expresión generadora dentro de la función `sum()` para filtrar y sumar únicamente los elementos pares.
20 – Contador de palabras	Contar cuántas palabras tiene un texto. Resultado en magenta.	**`texto.split()`**: Método que divide la cadena de texto en una lista de palabras, usando el espacio como separador. **`len(palabras)`**: Devuelve la cantidad de elementos (palabras) en la lista.

Ejercicios 21 al 30

Ejercicio	Enunciado y Código	Controles y Comandos Clave
21 – Número factorial	Calcular el factorial de un número usando un bucle. Resultado en cyan.	**`factorial *= i`**: Operador de asignación compuesta que acumula el producto, multiplicando el valor actual de `factorial` por `i` en cada iteración.
22 – Lista inversa	Invertir una lista de números y mostrarla en rojo.	**`numeros[::-1]`**: Slicing para invertir el orden de los elementos de la lista, similar a cómo se invierte un texto.
23 – Generador de números aleatorios	Generar 5 números aleatorios del 1 al 100 y mostrarlos en verde.	**`import random`**: Necesario para usar funciones de generación aleatoria. **`random.randint(1,100)`**: Genera un número entero aleatorio dentro del rango inclusivo.
24 – Palabras más largas	Mostrar la palabra más larga de un texto en azul.	**`max(palabras, key=len)`**: Función `max()` aplicada a la lista de palabras, utilizando `key=len` para determinar el elemento más largo (con la longitud máxima).
25 – Contador de letras y colores	Contar frecuencia de letras y aplicar color basado en la frecuencia (Azul: 1, Verde: 2, Amarillo: 3+).	**`contador.get(letra, 0) + 1`**: Este método de diccionario comprueba si la `letra` ya existe. Si no existe, devuelve 0 y luego suma 1; si existe, devuelve su valor actual y suma 1. Se usan condicionales anidadas para asignar colores.
26 – Guardar texto en archivo	Pedir texto al usuario y guardarlo en `texto.txt`. Confirmación en verde.	**`open("texto.txt", "w")`**: Abre el archivo `texto.txt` en modo escritura (`"w"`). **`archivo.write(texto)`**: Escribe el contenido en el archivo. **`with`**: Asegura que el archivo se cierre automáticamente al finalizar, incluso si ocurre un error.
27 – Leer archivo	Leer el contenido de `texto.txt` y mostrarlo en azul.	**`open("texto.txt", "r")`**: Abre el archivo en modo lectura (`"r"`). **`archivo.read()`**: Lee y devuelve todo el contenido del archivo como una única cadena de texto.
28 – Excepciones	Manejar errores de valor no numérico. Error en rojo, éxito en verde.	**`try/except`**: Estructura de manejo de errores que intenta ejecutar el código en `try` y, si falla, ejecuta el bloque `except`. **`ValueError`**: Error específico que ocurre al intentar convertir una

		cadena no numérica a entero (`int`).
29 – Lista de cuadrados usando función	Función que recibe una lista y devuelve otra con sus cuadrados. Resultado en cyan.	**`return [x**2 for x in lista]`**: La función utiliza una lista por comprensión para calcular los cuadrados y devuelve (`return`) la nueva lista.
30 – Contar letras en archivo	Leer `texto.txt` y contar cuántas veces aparece la letra "a" (sin importar mayúsculas/minúsculas). Resultado en amarillo.	**`contenido.lower().count("a")`**: Combina `.lower()` (convierte todo a minúsculas) y `.count("a")` para asegurar el conteo total de todas las apariciones de "a".

Ejercicios 31 al 40

Ejercicio	Enunciado y Código	Controles y Comandos Clave
31 – Diccionario de frecuencias desde archivo	Leer `texto.txt` y mostrar la frecuencia de cada letra con colores según la aparición (1=Azul, 2=Verde, 3+=Amarillo).	**Combinación de comandos**: Implica la lectura del archivo, la normalización del texto, la creación de un diccionario de frecuencias (usando `get(letra, 0) + 1`) y la asignación condicional de colores.
32 – Números pares y su suma	Mostrar 1-20 (pares en verde, impares en rojo) y mostrar la suma total de los pares en cyan.	**Acumulador `suma_pares`**: La variable `suma_pares` se inicializa en 0 y se incrementa (`+= i`) solo dentro del bloque `if` que detecta los números pares.
33 – Función para verificar palíndromo	Crear una función que devuelva `True` o `False` si un texto es palíndromo. Resultado en azul (True) o rojo (False).	**`return texto == texto[::-1]`**: La función realiza la normalización y luego devuelve directamente el resultado de la comparación lógica (que es un valor booleano `True` o `False`).
34 – Lista de números aleatorios y pares	Generar 10 números aleatorios (1-50). Mostrar Pares en verde, Impares en rojo, y la lista final de pares en azul.	**`pares.append(num)`**: Los números pares se recolectan en una lista separada (`pares`) mientras se itera sobre la lista original de aleatorios.
35 – Suma de números en archivo	Leer un archivo `numeros.txt` (con un número por línea), sumarlos y mostrar el resultado en amarillo.	**`archivo.readlines()`**: Devuelve una lista donde cada elemento es una línea del archivo. **`.strip()`**: Elimina cualquier espacio en blanco o salto de línea al inicio y final de cada línea. **`int(num.strip())`**: Convierte la línea limpia a número entero para poder sumarla.
36 – Promedio de números en archivo	Leer `numeros.txt`, calcular el promedio y mostrarlo en verde.	**Lista por comprensión para conversión**: `[int(linea.strip()) for linea in archivo]` convierte directamente cada línea a número entero. El promedio se calcula dividiendo `sum(numeros)` entre `len(numeros)`.
37 – Ordenar lista de números	Ordenar una lista de números de menor a mayor y mostrarla en azul.	**`numeros.sort()`**: Método que ordena la lista *in-place* (modifica la lista original) en orden ascendente.

38 – Diccionario de calificaciones	Mostrar aprobados (nota >= 5) en verde y suspensos en rojo.	**`if nota >= 5`**: Condicional simple para determinar el color de salida basado en el criterio de aprobación.
39 – Diccionario de palabras	Contar la frecuencia de cada palabra en un texto y mostrarlas en amarillo.	**`.lower().split()`**: Normaliza el texto a minúsculas y lo divide en palabras. Se utiliza el patrón de diccionario para contar la frecuencia de aparición de cada palabra.
40 – Juego: Adivina el número	Juego donde el usuario adivina un número aleatorio (1-10). Acierto en verde, fallo en rojo.	**`random.randint(1,10)`**: Genera el número secreto. El control se basa en una comparación directa `if intento == numero`.

Ejercicios 41 al 50

Ejercicio	Enunciado y Código	Controles y Comandos Clave
41 – Matriz y suma de filas	Crear una matriz (lista de listas) y mostrar la suma de cada fila en azul.	**`for fila in matriz`**: Itera sobre los elementos de la matriz, donde cada elemento es una lista que representa una fila. **`sum(fila)`**: Calcula la suma de los elementos de la fila actual.
42 – Diccionario anidado	Almacenar información de estudiantes en un diccionario anidado y mostrar notas con colores (aprobado/suspenso).	**Acceso anidado**: Se accede a los datos internos usando `datos["nota"]` para obtener la calificación dentro del diccionario anidado.
43 – Máximo y mínimo de lista	Mostrar el número mayor en verde y el menor en rojo de una lista.	**`max(lista)`**: Función que devuelve el valor máximo de la lista. **`min(lista)`**: Función que devuelve el valor mínimo de la lista.
44 – Filtrar números pares	Filtrar los números pares de una lista usando la función `filter`. Resultado en cyan.	**`filter(función, lista)`**: Función que construye un iterador con los elementos de la `lista` que satisfacen la condición de la `función`. **`lambda x: x%2==0`**: Función anónima utilizada para definir la condición de que el número sea par.
45 – Multiplicar todos los elementos	Multiplicar todos los elementos de una lista usando `reduce`. Resultado en amarillo.	**`from functools import reduce`**: La función `reduce` debe importarse del módulo `functools`. **`reduce(lambda x,y: x*y, numeros)`**: Aplica de manera acumulativa la función `lambda` (multiplicación) a todos los elementos de la lista.
46 – Archivos CSV	Crear un archivo `datos.csv` con nombres y edades. Confirmación en verde.	**`import csv`**: Necesario para trabajar con archivos CSV. **`csv.writer(archivo)`**: Crea un objeto escritor. **`escritor.writerow([...])`**: Escribe una fila de datos en el archivo CSV.
47 – Leer CSV	Leer `datos.csv` y mostrar cada fila en azul.	**`csv.reader(archivo)`**: Crea un objeto lector que puede iterar sobre las filas del archivo CSV.
48 – Manejo de excepciones múltiples	Pedir un número positivo y manejar dos errores: no numérico (rojo) o negativo (amarillo).	**`raise ValueError("Número negativo")`**: Permite lanzar una excepción personalizada si se cumple la condición de número negativo. El bloque `except ValueError` luego diferencia si el error es por "negativo" o por otro `ValueError` (no

		numérico) para asignar el color (rojo o amarillo).
49 – Juego: Piedra, Papel o Tijera	Implementar el juego contra la computadora. Mensajes en colores según el resultado.	**`random.choice(opciones)`**: Selecciona un elemento aleatorio de la lista para la elección de la CPU. El control se basa en una compleja estructura condicional (`if/elif/else`) que define las reglas de victoria.
50 – Calculadora básica	Implementar una calculadora (suma, resta, multiplicación, división). Cada operación tiene un color diferente. Manejo de división por cero.	**Estructura `if/elif`**: Se utiliza para seleccionar la operación basada en la entrada del usuario. **Manejo de División por Cero**: Se incluye un control específico (`if b != 0:`) para evitar errores matemáticos, mostrando un mensaje de error en rojo si se intenta dividir por cero.

Ejercicio 1 – Hola Mundo en color

Enunciado: Escribe un programa que muestre en pantalla "Hola Mundo" en color rojo.

Código con color:

```
# Importamos la librería para colores
from colorama import Fore, Style, init
init(autoreset=True)  # Inicializamos colorama para que se reinicie el color después de cada impresión

print(Fore.RED + "Hola Mundo")  # Muestra el texto en rojo
```

Explicación del código:

- `from colorama import Fore, Style, init`: Importa funciones de la librería `colorama` para usar colores y estilos.
- `init(autoreset=True)`: Inicializa la librería y reinicia automáticamente el color después de cada `print`.
- `print(Fore.RED + "Hola Mundo")`: `Fore.RED` cambia el color del texto a rojo.

Ejercicio 2 – Suma de dos números

Enunciado: Escribe un programa que pida dos números al usuario, los sume y muestre el resultado en azul.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

num1 = float(input("Introduce el primer número: "))  # Pide el primer número
num2 = float(input("Introduce el segundo número: "))  # Pide el segundo número

resultado = num1 + num2  # Suma los números

print(Fore.BLUE + "El resultado es:", resultado)  # Muestra el resultado en azul
```

Explicación del código:

- `float(input(...))`: Convierte la entrada del usuario en número decimal.
- `resultado = num1 + num2`: Suma los números.
- `print(Fore.BLUE + ...)`: Imprime el resultado en color azul.

Ejercicio 3 – Comparar dos números

Enunciado: Programa que reciba dos números y diga cuál es mayor. El mensaje se muestra en verde si el primer número es mayor, en amarillo si el segundo es mayor y en rojo si son iguales.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

a = int(input("Introduce el primer número: "))
b = int(input("Introduce el segundo número: "))

if a > b:
    print(Fore.GREEN + "El primer número es mayor")
elif b > a:
    print(Fore.YELLOW + "El segundo número es mayor")
else:
    print(Fore.RED + "Los números son iguales")
```

Explicación del código:

- `if/elif/else`: Condicionales que permiten ejecutar diferentes bloques según la condición.
- `Fore.GREEN`, `Fore.YELLOW`, `Fore.RED`: Cambian el color del texto según la condición.

Ejercicio 4 – Tabla de multiplicar

Enunciado: Programa que muestre la tabla de multiplicar de un número introducido por el usuario en color cyan.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

num = int(input("Introduce un número para la tabla: "))

for i in range(1, 11):  # Repite de 1 a 10
    print(Fore.CYAN + f"{num} x {i} = {num*i}")  # Muestra la multiplicación en
cyan
```

Explicación del código:

- `for i in range(1, 11)`: Bucle que va de 1 a 10.
- `f"{num} x {i} = {num*i}"`: Formato f-string para imprimir el resultado de la multiplicación.

Ejercicio 5 – Contador de letras

Enunciado: Programa que cuente cuántas veces aparece una letra en un texto. El resultado se muestra en magenta.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

texto = input("Introduce un texto: ")
letra = input("Introduce la letra a contar: ")

contador = texto.count(letra)  # Cuenta cuántas veces aparece la letra
print(Fore.MAGENTA + f"La letra '{letra}' aparece {contador} veces")
```

Explicación del código:

- `texto.count(letra)`: Método de strings que devuelve el número de veces que aparece `letra` en `texto`.
- `Fore.MAGENTA`: Muestra el resultado en magenta.

Ejercicio 6 – Números pares del 1 al 20

Enunciado: Programa que muestre todos los números pares del 1 al 20 en color verde.
Código con color:

```
from colorama import Fore, init
init(autoreset=True)

for i in range(1, 21):
    if i % 2 == 0:  # Comprueba si el número es divisible entre 2
        print(Fore.GREEN + str(i))  # Muestra el número en verde
```

Explicación del código:

- `i % 2 == 0`: Operador módulo `%` devuelve el resto de la división. Si es 0, es par.
- `str(i)`: Convierte el número en texto para poder aplicar el color.

Ejercicio 7 – Lista de nombres

Enunciado: Programa que pida 5 nombres y los muestre en color azul.
Código con color:

```
from colorama import Fore, init
init(autoreset=True)

nombres = []
for _ in range(5):
    nombre = input("Introduce un nombre: ")
    nombres.append(nombre)  # Añade el nombre a la lista

print(Fore.BLUE + "Nombres introducidos:", nombres)
```

Explicación del código:

- `nombres.append(nombre)`: Añade un elemento al final de la lista.
- `for _ in range(5)`: Repite 5 veces; _ se usa cuando no necesitamos la variable del bucle.

Ejercicio 8 – Suma de lista de números

Enunciado: Programa que sume todos los números de una lista y muestre el resultado en amarillo.
Código con color:

```
from colorama import Fore, init
init(autoreset=True)

numeros = [2, 5, 7, 10, 3]
total = sum(numeros)  # Suma todos los elementos de la lista
print(Fore.YELLOW + f"La suma es: {total}")
```

Explicación del código:

- `sum(numeros)`: Función que devuelve la suma de todos los elementos de la lista.

Ejercicio 9 – Número primo

Enunciado: Programa que determine si un número es primo. Resultado en cyan si es primo, rojo si no lo es.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

num = int(input("Introduce un número: "))
es_primo = True

if num < 2:
    es_primo = False
else:
    for i in range(2, int(num**0.5)+1):
        if num % i == 0:
            es_primo = False
            break

if es_primo:
    print(Fore.CYAN + f"{num} es primo")
else:
    print(Fore.RED + f"{num} no es primo")
```

Explicación del código:

- `int(num**0.5)+1`: Optimización para comprobar solo hasta la raíz cuadrada del número.
- `break`: Sale del bucle si encuentra un divisor.

Ejercicio 10 – Invertir texto

Enunciado: Programa que invierta un texto y lo muestre en magenta.
Código con color:

```
from colorama import Fore, init
init(autoreset=True)

texto = input("Introduce un texto: ")
invertido = texto[::-1]  # Invierte el texto usando slicing
print(Fore.MAGENTA + invertido)
```

Explicación del código:

- `texto[::-1]`: Slicing que recorre la cadena de atrás hacia adelante.

Ejercicio 11 – Contador de vocales

Enunciado: Programa que cuente cuántas vocales hay en un texto y muestre el resultado en verde.
Código con color:

```
from colorama import Fore, init
init(autoreset=True)


texto = input("Introduce un texto: ")
vocales = "aeiouAEIOU"
contador = sum(1 for letra in texto if letra in vocales)  # Cuenta vocales
print(Fore.GREEN + f"Hay {contador} vocales en el texto")
```

Explicación del código:

- `sum(1 for letra in texto if letra in vocales)`: Generador que cuenta cada letra que es vocal.

Ejercicio 12 – Lista al cuadrado

Enunciado: Programa que reciba una lista de números y muestre otra lista con los números al cuadrado en azul.

Código con color:

```
from colorama import Fore, init

init(autoreset=True)


numeros = [1, 2, 3, 4, 5]

cuadrados = [x**2 for x in numeros]  # Lista por comprensión con números al cuadrado

print(Fore.BLUE + "Números al cuadrado:", cuadrados)
```

Explicación del código:

- `[x**2 for x in numeros]`: Lista por comprensión que eleva cada número al cuadrado.

Ejercicio 13 – Media de números

Enunciado: Programa que calcule la media de una lista de números y muestre el resultado en amarillo.

Código con color:

```
from colorama import Fore, init

init(autoreset=True)


numeros = [5, 10, 15, 20]

media = sum(numeros)/len(numeros)  # Suma y divide entre la cantidad de elementos

print(Fore.YELLOW + f"La media es: {media}")
```

Explicación del código:

- `len(numeros)`: Devuelve el número de elementos de la lista.

Ejercicio 14 – Palíndromo

Enunciado: Programa que compruebe si un texto es palíndromo (se lee igual al derecho y al revés). Texto en cyan si lo es, rojo si no lo es.

Código con color:

```
from colorama import Fore, init

init(autoreset=True)


texto = input("Introduce un texto: ").replace(" ", "").lower()  # Ignora
espacios y mayúsculas

if texto == texto[::-1]:

    print(Fore.CYAN + "Es un palíndromo")

else:

    print(Fore.RED + "No es un palíndromo")
```

Explicación del código:

- `.replace(" ", "").lower()`: Normaliza el texto eliminando espacios y pasando a minúsculas.

Ejercicio 15 – Números impares y pares en colores

Enunciado: Programa que recorra números del 1 al 10 y muestre pares en verde e impares en rojo.
Código con color:

```
from colorama import Fore, init
init(autoreset=True)

for i in range(1, 11):
    if i % 2 == 0:
        print(Fore.GREEN + str(i))
    else:
        print(Fore.RED + str(i))
```

Explicación del código:

- Se combinan condicionales y colores para diferenciar pares e impares.

Ejercicio 16 – Función que saluda

Enunciado: Crea una función que reciba un nombre y muestre un saludo en color azul.
Código con color:

```
from colorama import Fore, init
init(autoreset=True)

def saludar(nombre):
    print(Fore.BLUE + f"Hola, {nombre}!")

nombre = input("Introduce tu nombre: ")
saludar(nombre)
```

Explicación del código:

- `def saludar(nombre):` Define una función que recibe un parámetro `nombre`.
- `f"Hola, {nombre}!"`: F-string para mostrar el saludo.
- `saludar(nombre)`: Llama a la función con el nombre introducido.

Ejercicio 17 – Números primos en lista

Enunciado: Programa que genere todos los números primos del 1 al 50 y los muestre en color cyan.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

primos = []
for num in range(2, 51):
    es_primo = True
    for i in range(2, int(num**0.5)+1):
        if num % i == 0:
            es_primo = False
            break
    if es_primo:
        primos.append(num)

print(Fore.CYAN + "Números primos del 1 al 50:", primos)
```

Explicación del código:

- `num**0.5`: Optimiza la comprobación solo hasta la raíz cuadrada del número.
- `primos.append(num)`: Añade el número primo a la lista.

Ejercicio 18 – Diccionario de edades

Enunciado: Crea un diccionario con nombres y edades, y muestra cada persona en verde si es mayor de 18, amarillo si no.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

edades = {"Ana": 20, "Luis": 16, "Maria": 25}

for nombre, edad in edades.items():
    if edad >= 18:
        print(Fore.GREEN + f"{nombre} tiene {edad} años")
    else:
        print(Fore.YELLOW + f"{nombre} tiene {edad} años")
```

Explicación del código:

- `for nombre, edad in edades.items()`: Recorre claves y valores del diccionario.
- `Fore.GREEN / Fore.YELLOW`: Cambia color según la edad.

Ejercicio 19 – Suma de números pares en lista

Enunciado: Programa que sume solo los números pares de una lista y muestre el resultado en azul.
Código con color:

```
from colorama import Fore, init
init(autoreset=True)

numeros = [1,2,3,4,5,6,7,8]
suma_pares = sum(x for x in numeros if x % 2 == 0)
print(Fore.BLUE + f"Suma de números pares: {suma_pares}")
```

Explicación del código:

- `sum(x for x in numeros if x % 2 == 0)`: Generador que suma solo los números pares.

Ejercicio 20 – Contador de palabras

Enunciado: Programa que cuente cuántas palabras tiene un texto y muestre el resultado en magenta.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

texto = input("Introduce un texto: ")
palabras = texto.split()  # Divide el texto en palabras
print(Fore.MAGENTA + f"El texto tiene {len(palabras)} palabras")
```

Explicación del código:

- `texto.split()`: Divide el texto en palabras usando espacios como separador.
- `len(palabras)`: Devuelve la cantidad de palabras.

Ejercicio 21 – Número factorial

Enunciado: Programa que calcule el factorial de un número usando bucle y muestre el resultado en cyan.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

num = int(input("Introduce un número: "))
factorial = 1
for i in range(1, num+1):
    factorial *= i  # Multiplica el factorial por cada número
print(Fore.CYAN + f"El factorial de {num} es {factorial}")
```

Explicación del código:

- `factorial *= i`: Acumula el producto de todos los números desde 1 hasta `num`.

Ejercicio 22 – Lista inversa

Enunciado: Programa que invierta una lista de números y la muestre en rojo.
Código con color:

```
from colorama import Fore, init
init(autoreset=True)

numeros = [1,2,3,4,5]
numeros_invertidos = numeros[::-1]  # Invierte la lista
print(Fore.RED + "Lista invertida:", numeros_invertidos)
```

Explicación del código:

- `numeros[::-1]`: Slicing que invierte la lista.

Ejercicio 23 – Generador de números aleatorios

Enunciado: Programa que genere 5 números aleatorios del 1 al 100 y los muestre en verde.
Código con color:

```
from colorama import Fore, init
import random
init(autoreset=True)

numeros = [random.randint(1,100) for _ in range(5)]
print(Fore.GREEN + "Números aleatorios:", numeros)
```

Explicación del código:

- `random.randint(1,100)`: Genera un número entero aleatorio entre 1 y 100.
- Lista por comprensión para repetir 5 veces.

Ejercicio 24 – Palabras más largas

Enunciado: Programa que muestre la palabra más larga de un texto en azul.
Código con color:

```
from colorama import Fore, init
init(autoreset=True)

texto = input("Introduce un texto: ")
palabras = texto.split()
mas_larga = max(palabras, key=len)  # Encuentra la palabra más larga
print(Fore.BLUE + f"La palabra más larga es: {mas_larga}")
```

Explicación del código:

- `max(palabras, key=len)`: Devuelve el elemento con mayor longitud de la lista.

Ejercicio 25 – Contador de letras y colores

Enunciado: Programa que cuente cuántas veces aparece cada letra de un texto y muestre las letras en colores diferentes según su frecuencia:

- Azul si aparece 1 vez, verde 2 veces, amarillo 3 o más veces.
  Código con color:

```
from colorama import Fore, init
init(autoreset=True)

texto = input("Introduce un texto: ").replace(" ", "").lower()
contador = {}

for letra in texto:
    contador[letra] = contador.get(letra, 0) + 1

for letra, veces in contador.items():
    if veces == 1:
        color = Fore.BLUE
    elif veces == 2:
        color = Fore.GREEN
    else:
        color = Fore.YELLOW
    print(color + f"{letra}: {veces}")
```

Explicación del código:

- `contador.get(letra, 0) + 1`: Si la letra no existe en el diccionario, devuelve 0 y suma 1.
- Condicionales asignan colores según frecuencia.

Ejercicio 26 – Guardar texto en archivo

Enunciado: Programa que pida un texto al usuario y lo guarde en un archivo llamado `texto.txt`. El mensaje de confirmación se muestra en verde.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

texto = input("Introduce un texto para guardar en archivo: ")

with open("texto.txt", "w") as archivo:
    archivo.write(texto)  # Escribe el texto en el archivo

print(Fore.GREEN + "Texto guardado correctamente en texto.txt")
```

Explicación del código:

- `open("texto.txt", "w")`: Abre el archivo en modo escritura.
- `archivo.write(texto)`: Escribe el texto en el archivo.
- `with`: Garantiza que el archivo se cierre automáticamente al terminar.

Ejercicio 27 – Leer archivo

Enunciado: Programa que lea el contenido de `texto.txt` y lo muestre en azul.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

with open("texto.txt", "r") as archivo:
    contenido = archivo.read()  # Lee todo el contenido del archivo

print(Fore.BLUE + "Contenido del archivo:")
print(Fore.BLUE + contenido)
```

Explicación del código:

- `archivo.read()`: Devuelve todo el contenido del archivo como cadena.

Ejercicio 28 – Excepciones

Enunciado: Programa que pida un número y maneje el error si el usuario introduce un valor no numérico. Mensajes en rojo si hay error y en verde si es correcto.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

try:
    num = int(input("Introduce un número: "))
    print(Fore.GREEN + f"Has introducido el número {num}")
except ValueError:
    print(Fore.RED + "Error: Debes introducir un número válido")
```

Explicación del código:

- `try/except`: Permite capturar errores y manejarlo sin que el programa falle.
- `ValueError`: Error que ocurre al convertir texto no numérico a entero.

Ejercicio 29 – Lista de cuadrados usando función

Enunciado: Crea una función que reciba una lista de números y devuelva otra con sus cuadrados. Resultado en cyan.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

def cuadrados(lista):
    return [x**2 for x in lista]

numeros = [1,2,3,4,5]
resultado = cuadrados(numeros)
print(Fore.CYAN + "Cuadrados:", resultado)
```

Explicación del código:

- `return [x**2 for x in lista]`: Devuelve una nueva lista con cada número al cuadrado.

Ejercicio 30 – Contar letras en archivo

Enunciado: Programa que lea `texto.txt` y cuente cuántas veces aparece la letra "a". Resultado en amarillo.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

with open("texto.txt", "r") as archivo:
    contenido = archivo.read()

contador = contenido.lower().count("a")  # Convierte a minúsculas y cuenta la letra a
print(Fore.YELLOW + f"La letra 'a' aparece {contador} veces")
```

Explicación del código:

- `.lower()`: Convierte todo el texto a minúsculas para contar todas las "a".

Ejercicio 31 – Diccionario de frecuencias desde archivo

Enunciado: Programa que lea `texto.txt` y muestre la frecuencia de cada letra en colores diferentes: azul si aparece 1 vez, verde 2 veces, amarillo 3 o más veces.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

with open("texto.txt", "r") as archivo:
    texto = archivo.read().replace(" ", "").lower()

contador = {}
for letra in texto:
    contador[letra] = contador.get(letra, 0) + 1

for letra, veces in contador.items():
    if veces == 1:
        color = Fore.BLUE
    elif veces == 2:
        color = Fore.GREEN
    else:
        color = Fore.YELLOW
    print(color + f"{letra}: {veces}")
```

Explicación del código:

- Se combina lectura de archivo, diccionario y colores según frecuencia de cada letra.

Ejercicio 32 – Números pares y su suma

Enunciado: Programa que genere números del 1 al 20, los muestre en verde si son pares, rojo si impares, y al final muestre la suma de los pares en cyan.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

suma_pares = 0
for i in range(1, 21):
    if i % 2 == 0:
        print(Fore.GREEN + str(i))
        suma_pares += i
    else:
        print(Fore.RED + str(i))

print(Fore.CYAN + f"Suma de los pares: {suma_pares}")
```

Explicación del código:

- Se usan condicionales para diferenciar pares e impares y acumular suma.

Ejercicio 33 – Función para verificar palíndromo

Enunciado: Crea una función que reciba un texto y devuelva True si es palíndromo, False si no. Resultado en azul si es palíndromo, rojo si no.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

def es_palindromo(texto):
    texto = texto.replace(" ", "").lower()
    return texto == texto[::-1]

texto = input("Introduce un texto: ")
if es_palindromo(texto):
    print(Fore.BLUE + "Es un palíndromo")
else:
    print(Fore.RED + "No es un palíndromo")
```

Explicación del código:

- `return texto == texto[::-1]`: Devuelve True si el texto es igual a su inverso.

Ejercicio 34 – Lista de números aleatorios y pares

Enunciado: Programa que genere 10 números aleatorios del 1 al 50, muestre pares en verde, impares en rojo y al final los pares en azul.

Código con color:

```
from colorama import Fore, init
import random
init(autoreset=True)

numeros = [random.randint(1,50) for _ in range(10)]
pares = []

for num in numeros:
    if num % 2 == 0:
        print(Fore.GREEN + str(num))
        pares.append(num)
    else:
        print(Fore.RED + str(num))

print(Fore.BLUE + "Números pares:", pares)
```

Explicación del código:

- Combinación de números aleatorios, bucles, condicionales y listas para pares.

Ejercicio 35 – Suma de números en archivo

Enunciado: Programa que lea un archivo `numeros.txt` con un número en cada línea, sume todos los números y muestre el resultado en amarillo.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

with open("numeros.txt", "r") as archivo:
    numeros = archivo.readlines()  # Lee todas las líneas

suma = sum(int(num.strip()) for num in numeros)  # Convierte a entero y suma
print(Fore.YELLOW + f"Suma de los números: {suma}")
```

Explicación del código:

- `.readlines()`: Devuelve una lista con cada línea del archivo.
- `.strip()`: Elimina espacios o saltos de línea.
- `int(...)`: Convierte cada línea en número entero.

Ejercicio 36 – Promedio de números en archivo

Enunciado: Programa que lea un archivo `numeros.txt`, calcule el promedio y lo muestre en verde.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

with open("numeros.txt", "r") as archivo:
    numeros = [int(linea.strip()) for linea in archivo]

promedio = sum(numeros) / len(numeros)
print(Fore.GREEN + f"El promedio es: {promedio}")
```

Explicación:

- Lista por comprensión para convertir cada línea en número.
- Promedio = suma ÷ cantidad de elementos.

Ejercicio 37 – Ordenar lista de números

Enunciado: Programa que ordene una lista de números de menor a mayor y los muestre en azul.
Código con color:

```
from colorama import Fore, init
init(autoreset=True)

numeros = [15, 3, 9, 20, 1, 7]
numeros.sort()
print(Fore.BLUE + f"Números ordenados: {numeros}")
```

Explicación:

- `.sort()`: Ordena la lista en orden ascendente.

Ejercicio 38 – Diccionario de calificaciones

Enunciado: Programa que guarde nombres y calificaciones en un diccionario y muestre los aprobados en verde y suspensos en rojo.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

calificaciones = {"Ana": 8, "Luis": 4, "Marta": 6, "Pedro": 3}

for nombre, nota in calificaciones.items():
    if nota >= 5:
        print(Fore.GREEN + f"{nombre} ha aprobado con {nota}")
    else:
        print(Fore.RED + f"{nombre} ha suspendido con {nota}")
```

Explicación:

- `diccionario.items()`: Recorre clave y valor.

Ejercicio 39 – Diccionario de palabras

Enunciado: Programa que cuente cuántas veces aparece cada palabra en un texto y las muestre en amarillo.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

texto = input("Introduce un texto: ").lower().split()
contador = {}

for palabra in texto:
    contador[palabra] = contador.get(palabra, 0) + 1

for palabra, veces in contador.items():
    print(Fore.YELLOW + f"{palabra}: {veces}")
```

Explicación:

- `.split()`: Separa el texto en palabras.
- Diccionario cuenta frecuencia.

Ejercicio 40 – Juego: Adivina el número

Enunciado: Programa que genere un número aleatorio del 1 al 10 y el usuario debe adivinarlo. Verde si acierta, rojo si falla.

Código con color:

```
from colorama import Fore, init
import random
init(autoreset=True)

numero = random.randint(1,10)
intento = int(input("Adivina el número (1-10): "))

if intento == numero:
    print(Fore.GREEN + "¡Correcto!")
else:
    print(Fore.RED + f"Fallaste, el número era {numero}")
```

Explicación:

- `random.randint(1,10)`: Genera número aleatorio entre 1 y 10.

Ejercicio 41 – Matriz y suma de filas

Enunciado: Programa que cree una matriz (lista de listas) y muestre la suma de cada fila en azul.
Código con color:

```
from colorama import Fore, init
init(autoreset=True)

matriz = [[1,2,3],[4,5,6],[7,8,9]]

for fila in matriz:
    print(Fore.BLUE + f"Suma de {fila} = {sum(fila)}")
```

Explicación:

- Cada fila es una lista dentro de la matriz.
- `sum(fila)`: Suma de cada fila.

Ejercicio 42 – Diccionario anidado

Enunciado: Programa que guarde información de estudiantes en un diccionario anidado y muestre notas en colores.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

estudiantes = {
    "Ana": {"edad": 20, "nota": 9},
    "Luis": {"edad": 22, "nota": 4}
}

for nombre, datos in estudiantes.items():
    nota = datos["nota"]
    if nota >= 5:
        print(Fore.GREEN + f"{nombre}, edad {datos['edad']}, aprobado con
{nota}")
    else:
        print(Fore.RED + f"{nombre}, edad {datos['edad']}, suspendido con
{nota}")
```

Explicación:

- Diccionarios anidados permiten almacenar varias propiedades.

Ejercicio 43 – Máximo y mínimo de lista

Enunciado: Programa que muestre el número mayor en verde y el menor en rojo de una lista.
Código con color:

```
from colorama import Fore, init
init(autoreset=True)

numeros = [3,7,1,9,4,8]
print(Fore.GREEN + f"Máximo: {max(numeros)}")
print(Fore.RED + f"Mínimo: {min(numeros)}")
```

Explicación:

- `max(lista)`: Devuelve el valor máximo.
- `min(lista)`: Devuelve el valor mínimo.

Ejercicio 44 – Filtrar números pares

Enunciado: Programa que filtre los números pares de una lista usando `filter` y los muestre en cyan.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

numeros = [1,2,3,4,5,6,7,8]
pares = list(filter(lambda x: x%2==0, numeros))
print(Fore.CYAN + f"Números pares: {pares}")
```

Explicación:

- `filter(función, lista)`: Devuelve los elementos que cumplen la condición.
- `lambda`: Función anónima.

Ejercicio 45 – Multiplicar todos los elementos

Enunciado: Programa que multiplique todos los elementos de una lista usando `reduce`. Resultado en amarillo.

Código con color:

```
from colorama import Fore, init
from functools import reduce
init(autoreset=True)

numeros = [1,2,3,4,5]
producto = reduce(lambda x,y: x*y, numeros)
print(Fore.YELLOW + f"Producto: {producto}")
```

Explicación:

- `reduce`: Aplica una función acumulativa a la lista.
- Multiplica todos los elementos.

Ejercicio 46 – Archivos CSV

Enunciado: Programa que cree un archivo `datos.csv` con nombres y edades, y muestre confirmación en verde.

Código con color:

```
from colorama import Fore, init
import csv
init(autoreset=True)

with open("datos.csv", "w", newline="") as archivo:
    escritor = csv.writer(archivo)
    escritor.writerow(["Nombre", "Edad"])
    escritor.writerow(["Ana", 20])
    escritor.writerow(["Luis", 25])

print(Fore.GREEN + "Archivo datos.csv creado con éxito")
```

Explicación:

- `csv.writer`: Escribe datos en formato CSV.
- `.writerow`: Escribe una fila.

Ejercicio 47 – Leer CSV

Enunciado: Programa que lea `datos.csv` y muestre cada fila en azul.

Código con color:

```
from colorama import Fore, init
import csv
init(autoreset=True)

with open("datos.csv", "r") as archivo:
    lector = csv.reader(archivo)
    for fila in lector:
        print(Fore.BLUE + str(fila))
```

Explicación:

- `csv.reader`: Permite leer un archivo CSV línea por línea.

Ejercicio 48 – Manejo de excepciones múltiples

Enunciado: Programa que pida un número y maneje dos errores: si no es número (rojo) o si es negativo (amarillo).

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

try:
    num = int(input("Introduce un número positivo: "))
    if num < 0:
        raise ValueError("Número negativo")
    print(Fore.GREEN + f"Número válido: {num}")
except ValueError as e:
    if "negativo" in str(e).lower():
        print(Fore.YELLOW + "Error: No se permiten números negativos")
    else:
        print(Fore.RED + "Error: Debes introducir un número válido")
```

Explicación:

- `raise`: Lanza una excepción personalizada.
- Maneja diferentes errores con colores distintos.

Ejercicio 49 – Juego: Piedra, Papel o Tijera

Enunciado: Programa de piedra, papel o tijera contra la computadora. Mensajes en colores según resultado.

Código con color:

```
from colorama import Fore, init
import random
init(autoreset=True)

opciones = ["piedra", "papel", "tijera"]
jugador = input("Elige piedra, papel o tijera: ").lower()
cpu = random.choice(opciones)

print(Fore.CYAN + f"La computadora eligió {cpu}")

if jugador == cpu:
    print(Fore.YELLOW + "Empate")
elif (jugador == "piedra" and cpu == "tijera") or \
     (jugador == "papel" and cpu == "piedra") or \
     (jugador == "tijera" and cpu == "papel"):
    print(Fore.GREEN + "¡Ganaste!")
else:
    print(Fore.RED + "Perdiste")
```

Explicación:

- `random.choice(lista)`: Selecciona un elemento aleatorio de la lista.

Ejercicio 50 – Calculadora básica

Enunciado: Programa que implemente una calculadora básica (suma, resta, multiplicación y división). Resultado en colores diferentes según operación.

Código con color:

```
from colorama import Fore, init
init(autoreset=True)

a = float(input("Introduce el primer número: "))
b = float(input("Introduce el segundo número: "))
operacion = input("Elige operación (+, -, *, /): ")

if operacion == "+":
    print(Fore.GREEN + f"Resultado: {a+b}")
elif operacion == "-":
    print(Fore.YELLOW + f"Resultado: {a-b}")
elif operacion == "*":
    print(Fore.BLUE + f"Resultado: {a*b}")
elif operacion == "/":
    if b != 0:
        print(Fore.CYAN + f"Resultado: {a/b}")
    else:
        print(Fore.RED + "Error: No se puede dividir entre 0")
else:
    print(Fore.RED + "Operación no válida")
```

Explicación:

- Condicionales eligen la operación.
- Cada operación tiene un color diferente.
- Se maneja la división entre 0 con un error en rojo.

MANUAL DETALLADO DE JUEGOS Y EJERCICIOS

Juego 1: Adivina el número

Enunciado: El programa elige un número aleatorio entre 1 y 10, y el jugador debe adivinarlo.

Comando / Control del Jugador	Función
`int(input("..."))`	Pide al jugador que introduzca un número y se asegura de que este se guarde como un valor **entero**.

Explicación Detallada de Comandos y Lógica:

1. **`import random`**: Se utiliza esta instrucción para importar la librería que permite la generación de números al azar.
2. **`random.randint(1, 10)`**: Genera el número secreto, que será un **número aleatorio** entre 1 y 10 (incluidos).
3. **`if intento == numero`**: Esta condición es la que se usa para **comparar** el número introducido por el jugador (`intento`) con el número secreto (`numero`) y determinar si ha acertado.
4. **`print(...)`**: Muestra los mensajes de resultado ("¡Acertaste!" o "Fallaste") en pantalla.

Juego 2: Piedra, papel o tijera

Enunciado: El usuario juega el clásico juego de piedra, papel o tijera contra la computadora.

Comando / Control del Jugador	Función
`input("Elige piedra, papel o tijera: ")`	Pide la jugada deseada al usuario. El jugador debe escribir exactamente "piedra", "papel" o "tijera".

Explicación Detallada de Comandos y Lógica:

1. **`opciones = [...]`**: Define una lista de jugadas posibles.
2. **`random.choice(opciones)`**: La computadora elige su jugada al **azar** de la lista de opciones.
3. **`if, elif, else`**: Estas estructuras de control son las que **deciden el ganador** basándose en las reglas del juego (por ejemplo, `usuario == "piedra" and pc == "tijera"`).
4. **`print("La computadora eligió:", pc)`**: Muestra el resultado de la elección de la computadora.

Juego 3: Adivina la palabra

Enunciado: El jugador debe adivinar la palabra secreta ("python") y tiene solo **tres intentos**.

Comando / Control del Jugador	Función
`input("Adivina la palabra secreta: ")`	Permite al jugador ingresar su respuesta.

Explicación Detallada de Comandos y Lógica:

1. **`palabra = "python"`** / **`intentos = 3`**: Establecen la palabra secreta y el número de intentos permitidos.
2. **`while intentos > 0`**: Este es un **bucle** que se repite y mantiene el juego activo mientras el número de intentos restantes sea mayor que cero.
3. **`if respuesta == palabra`**: Verifica si la respuesta ingresada coincide con la palabra secreta.
4. **`break`**: Si el jugador acierta, este comando **termina inmediatamente** el bucle `while`.
5. **`intentos -= 1`**: Si el jugador falla, esta instrucción **resta uno** al contador de intentos restantes.

Juego 4: Lanzar un dado

Enunciado: El programa simula el lanzamiento de un dado estándar de seis caras.

Comandos / Controles: Este es un ejercicio de simulación. El jugador solo observa el resultado.

Explicación Detallada de Comandos y Lógica:

1. **`random.randint(1, 6)`**: Genera un **número aleatorio** que representa el resultado del dado, entre 1 y 6.
2. **`print("El dado muestra:", dado)`**: Muestra el número generado en la pantalla.

Juego 5: Adivina el número con pistas

Enunciado: El jugador debe adivinar un número entre 1 y 20, recibiendo pistas sobre si su intento es **mayor** o **menor** que el número secreto.

Comando / Control del Jugador	Función
`int(input("..."))`	Pide al jugador que introduzca su intento numérico entre 1 y 20.

Explicación Detallada de Comandos y Lógica:

1. **`while True`**: Crea un **bucle infinito**. El juego solo terminará (se detendrá) cuando el jugador acierte.
2. **`if intento == numero`**: Condición que verifica el acierto y es seguida por `break` para terminar el bucle.
3. **`elif intento < numero`**: Si el intento es menor, el programa da la **pista** indicando que el número secreto es **mayor**.
4. **`else`**: Si no es correcto y no es menor, la única opción restante es que sea mayor, dando la **pista** de que el número secreto es **menor**.

Juego 6: Cara o cruz

Enunciado: Simula el lanzamiento de una moneda.

Comandos / Controles: Este es un ejercicio de simulación. El jugador solo observa el resultado.

Explicación Detallada de Comandos y Lógica:

1. **`random.choice(["cara", "cruz"])`**: Elige al **azar** una de las dos opciones ("cara" o "cruz") de la lista.
2. **`print(...)`**: Muestra en pantalla el resultado en el que cayó la moneda.

Juego 7: Ahorcado simple

Enunciado: El jugador debe adivinar una palabra secreta ("gato") letra por letra.

Comando / Control del Jugador	Función
`input("Adivina una letra: ")`	Pide al jugador que ingrese una única letra en cada turno.
`" ".join(oculto)`	Muestra el estado actual de la palabra, con las letras adivinadas separadas por espacios.

Explicación Detallada de Comandos y Lógica:

1. **`oculto = ["_"] * len(palabra)`**: Crea la representación inicial de la palabra oculta como una lista de guiones bajos.
2. **`while "_" in oculto`**: El juego continúa mientras exista al menos un guion bajo en la lista `oculto` (es decir, mientras queden letras por adivinar).
3. **`if letra in palabra`**: Comprueba si la letra introducida por el jugador se encuentra en la palabra secreta.
4. **`for i in range(len(palabra))`**: Itera sobre los **índices** de la palabra para encontrar dónde reemplazar el guion bajo por la letra correcta si esta fue acertada.

Juego 8: Número par o impar

Enunciado: La computadora elige un número entre 1 y 100, y el jugador debe adivinar si es par o impar.

Comando / Control del Jugador	Función
`input("¿Es par o impar?: ")`	Pide al jugador que escriba su respuesta ("par" o "impar").

Explicación Detallada de Comandos y Lógica:

1. **`numero = random.randint(1, 100)`**: Genera el número secreto.
2. **`numero % 2 == 0`**: Esta es la operación clave para **verificar la paridad**. El operador `%` (módulo) devuelve el resto de una división. Si el resto al dividir el número entre 2 es 0, el número es **par**.
3. **`if, elif, else`**: Evalúan la combinación de la paridad real del número con la respuesta del usuario para determinar si es correcto o incorrecto.

Juego 9: Adivina el número en intentos limitados

Enunciado: El jugador tiene **5 intentos** para adivinar un número entre 1 y 50.

Comando / Control del Jugador	Función
`int(input("Adivina un número: "))`	Pide al jugador que introduzca su intento numérico.

Explicación Detallada de Comandos y Lógica:

1. **`intentos = 5`**: Establece el **máximo de intentos** permitidos.
2. **`while intentos > 0`**: Define un **bucle limitado** que se repite mientras queden intentos.
3. **`elif guess < numero`**: Da la pista de que el intento es **"Demasiado bajo"**.
4. **`else`**: Da la pista de que el intento es **"Demasiado alto"**.
5. **`intentos -= 1`**: Se ejecuta cada vez que el jugador falla, reduciendo el contador de intentos.
6. **`if intentos == 0`**: Se verifica al final del bucle para notificar al jugador si perdió por agotar los intentos.

Juego 10: Juego de sumas

Enunciado: El programa presenta una suma de dos números aleatorios (entre 1 y 10) que el jugador debe resolver.

Comando / Control del Jugador	Función
`int(input(f"¿Cuánto es {a} + {b}?: "))`	El jugador introduce la respuesta numérica de la suma.

Explicación Detallada de Comandos y Lógica:

1. **`random.randint(1, 10)`**: Se usa para generar los dos números (`a` y `b`) que se van a sumar.
2. **`f"..."` (f-string)**: Permite **insertar las variables** (`a` y `b`) directamente en el texto de la pregunta mostrada al jugador.
3. **`a + b`**: Calcula la respuesta correcta de la suma.
4. **`if respuesta == a + b`**: Compara la respuesta introducida por el jugador con el resultado matemático correcto.

Juego 11: Adivina el color

Enunciado: La computadora elige un color de una lista, y el jugador debe adivinar cuál es.

Comando / Control del Jugador	Función
`input("Adivina un color (rojo, azul, verde, amarillo): ")`	El jugador ingresa el color que cree que es el secreto.

Explicación Detallada de Comandos y Lógica:

1. **`colores = [...]`**: Define la lista de posibles opciones de color.
2. **`random.choice(colores)`**: Elige el color **secreto** de manera aleatoria.
3. La comparación final se hace con una estructura `if/else`.

Juego 12: Suma de dados

Enunciado: Se simula el lanzamiento de dos dados. El jugador **gana si la suma de los valores es 7**.

Comandos / Controles: Este es un ejercicio de simulación. El jugador observa el resultado de la suma.

Explicación Detallada de Comandos y Lógica:

1. **`dado1 = random.randint(1, 6)`** y **`dado2 = random.randint(1, 6)`**: Generan los valores individuales de los dos dados.
2. **`suma = dado1 + dado2`**: Calcula el valor total de la suma de ambos dados.
3. **`if suma == 7`**: Esta es la condición de victoria para el juego.

Juego 13: Cuenta atrás

Enunciado: Juego simple que cuenta del 5 al 1 antes de mostrar "¡Despegue!".

Comandos / Controles: El jugador observa la secuencia de conteo.

Explicación Detallada de Comandos y Lógica:

1. **`import time`**: Importa la librería necesaria para manejar pausas.
2. **`for i in range(5, 0, -1)`**: Crea un **bucle decreciente** que empieza en 5, termina en 1 (el 0 es excluido) y avanza en pasos de -1.
3. **`time.sleep(1)`**: Genera una **pausa de un segundo** después de mostrar cada número.

Juego 14: Mayor de dos números

Enunciado: El jugador introduce dos números, y el programa determina cuál es el mayor.

Comando / Control del Jugador	Función
`int(input("Escribe el... número: "))`	Pide al jugador que introduzca los dos números (A y B).

Explicación Detallada de Comandos y Lógica:

1. **`if a > b`**: Compara si el primer número es mayor que el segundo.
2. **`elif b > a`**: Es la segunda comparación, verifica si B es mayor que A.
3. **`else`**: Este caso se ejecuta si las dos condiciones anteriores son falsas, lo que implica que los números **son iguales**.

Juego 15: Tabla de multiplicar

Enunciado: El jugador introduce un número, y el programa muestra su tabla de multiplicar hasta el 10.

Comando / Control del Jugador	Función
`int(input("Introduce un número: "))`	El jugador introduce el número base (`n`) para la tabla.

Explicación Detallada de Comandos y Lógica:

1. **`for i in range(1, 11)`**: Crea un **bucle** que itera diez veces, con la variable `i` tomando valores desde 1 hasta 10.
2. **`print(n, "x", i, "=", n * i)`**: Dentro del bucle, muestra cada operación de multiplicación, usando `n * i` para calcular el resultado.

Juego 16: Adivina la letra

Enunciado: El jugador debe adivinar una letra secreta elegida al azar del rango "a" a "g".

Comando / Control del Jugador	Función
`input("Adivina una letra (a-g): ")`	El jugador ingresa su letra de elección.

Explicación Detallada de Comandos y Lógica:

1. **`letras = "abcdefg"`**: Define la cadena de opciones posibles para la letra secreta.
2. **`random.choice(letras)`**: Elige una letra **al azar** de la cadena de opciones.
3. La comparación final se realiza para determinar si la elección del jugador es correcta o incorrecta.

Juego 17: Juego del conteo

Enunciado: El jugador escribe un número, y el programa cuenta desde 1 hasta ese número.

Comando / Control del Jugador	Función
`int(input("Escribe un número: "))`	El jugador introduce el número final (`n`) del conteo.

Explicación Detallada de Comandos y Lógica:

1. **`for i in range(1, n + 1)`**: Crea un **bucle** que cuenta de 1 hasta el valor de `n`. Se usa `n + 1` porque la función `range` excluye el número final, asegurando que `n` sea contado.
2. **`print(i)`**: Muestra el número actual en el conteo en cada iteración.

Juego 18: Número secreto con rango

Enunciado: El jugador debe adivinar un número entre 1 y 100. El programa proporciona pistas de "Muy bajo" o "Muy alto".

Comando / Control del Jugador	Función
`int(input("Adivina el número: "))`	El jugador introduce su intento.

Explicación Detallada de Comandos y Lógica:

1. **`secreto = random.randint(1, 100)`**: Genera el número secreto entre 1 y 100.
2. **`while True`**: Bucle infinito que mantiene el juego activo hasta que el jugador acierta.
3. **`if intento == secreto`**: Condición de acierto seguida por `break` para terminar.
4. **`elif intento < secreto`**: Pista que indica que el intento fue "**Muy bajo**".

5. **`else`**: Pista que indica que el intento fue "**Muy alto**".

Juego 19: Juego del cuadrado

Enunciado: El jugador debe responder cuál es el cuadrado de un número aleatorio entre 1 y 10.

Comando / Control del Jugador	Función
`int(input(f"..."))`	El jugador introduce la respuesta numérica.

Explicación Detallada de Comandos y Lógica:

1. **`n = random.randint(1, 10)`**: Genera el número base.
2. **`n ** 2`**: Calcula el **cuadrado** del número (`n`) utilizando el operador de exponenciación.
3. **`if respuesta == n ** 2`**: Compara la respuesta del jugador con el cálculo correcto.

Juego 20: Número mayor de tres

Enunciado: El jugador introduce tres números, y el programa determina cuál de ellos es el mayor.

Comando / Control del Jugador	Función
`int(input("Primer número: "))` / **`int(input("Segundo número: "))`** / **`int(input("Tercer número: "))`**	El jugador introduce los tres números, `a`, `b`, y `c`.

Explicación Detallada de Comandos y Lógica:

1. **`if a > b and a > c`**: Esta condición usa el operador lógico **`and`** para comprobar si el número `a` es mayor que `b` **Y** a la vez mayor que `c`.
2. **`elif b > a and b > c`**: Si la primera condición falla, esta comprueba si `b` es mayor que `a` **Y** a la vez mayor que `c`.
3. **`else`**: Si ni `a` ni `b` cumplen las condiciones de ser el mayor, el `else` determina que `c` es el mayor (o que son iguales, pero se designa a `c` en este caso).

MANUAL EXPLICATIVO DE CONTROLES Y COMANDOS BÁSICOS

(Basado en Fuentes de Caracteres Especiales, Control de Consola y Estructuras de Control)

Este manual detalla los controles y comandos asociados con la gestión de texto, formato de salida y estructuras básicas de programación según los extractos proporcionados.

1. Caracteres Especiales y Secuencias de Escape

Estos comandos, utilizados generalmente dentro de la función `print()`, permiten controlar el formato y la presentación del texto. Se dividen en controles de formato, controles de texto/sonido y controles literales.

1.1 Controles de Salto de Línea y Espaciado

Carácter	Significado	Explicación Detallada	Comando de Ejemplo	Salida Resultante
\n	Nueva línea	Inserta un salto de línea, moviendo el texto subsiguiente al inicio de la siguiente línea.	`print("Hola \nMundo")`	Hola seguido de Mundo en una nueva línea.
\t	Tabulación horizontal	Inserta un espacio de tabulación, útil para alinear columnas de datos.	`print("Col1 \tCol2\tCol 3")`	Col1, Col2 y Col3 espaciados por tabulaciones.
\r	Retorno de carro	Mueve el cursor al inicio de la línea actual. El texto que sigue a `\r` sobrescribe los caracteres iniciales de esa línea.	`print("1234 5\rAB")`	AB345 (AB sobrescribe 12).
\v	Tabulación vertical	Inserta un espacio de tabulación vertical entre las líneas de texto.	`print("Line a1\vLinea 2")`	Linea1 y Linea2 aparecerán con un espaciado vertical entre ellas.
\f	Salto de página	Inserta un carácter de salto de página.	`print("Line a1\fLinea 2")`	Linea1 y Linea2 separadas por un salto de página.

1.2 Controles Literales (Comillas y Barra Invertida)

Estos controles son esenciales cuando se desea imprimir caracteres que, de otro modo, el lenguaje interpretaría como comandos de delimitación de *string*.

Carácter	Significado	Explicación Detallada	Comando de Ejemplo	Salida Resultante
'	Comilla simple	Permite incluir una comilla simple dentro de una cadena delimitada por comillas simples.	`print('It\'s Python')`	It's Python.
"	Comilla doble	Permite incluir una comilla doble dentro de una cadena delimitada por comillas dobles.	`print("Ella dijo: \"Hola\"")`	Ella dijo: "Hola".

\	Barra invertida literal	Permite imprimir el carácter de barra invertida (\) sin que se interprete como el inicio de una secuencia de escape.	`print("C:\\Usuarios\\Python")`	C:\Usuarios\Python.

1.3 Controles de Texto, Sonido y Borrado

Carácter	Significado	Explicación Detallada	Comando de Ejemplo	Salida Resultante
\a	Sonido de alerta	Produce un sonido audible (campana o alerta) en la consola o sistema operativo.	`print("\a")`	Se emite el sonido de alerta.
\b	Retroceso (Backspace)	Borra el carácter inmediatamente anterior en la salida.	`print("Hola\bMundo")`	HolMundo (La 'a' es borrada por \b).
\0	Carácter nulo	Inserta un carácter nulo, el cual generalmente no es visible.	`print("A\0B")`	AB (con un carácter nulo entre A y B).

1.4 Controles Unicode y Hexadecimales

Estos controles permiten representar caracteres utilizando sus códigos numéricos.

Carácter	Significado	Explicación Detallada	Comando de Ejemplo	Salida Resultante
\uXXXX	Carácter Unicode (4 dígitos hex)	Representa un carácter Unicode utilizando cuatro dígitos hexadecimales.	`print("\u2764")`	Se muestra el carácter ❤.
\UXXXXXXXX	Carácter Unicode (8 dígitos hex)	Representa un carácter Unicode utilizando ocho dígitos hexadecimales.	`print("\U0001F600")`	Se muestra el carácter 😀.
\xXX	Carácter hexadecimal	Representa un carácter utilizando dos dígitos hexadecimales.	`print("\x41")`	A (ya que 41 es el código ASCII/hexadecimal para 'A').

---

2. Strings Crudos (Raw Strings)

El control de *raw strings* se utiliza cuando se desea **ignorar la interpretación de los caracteres especiales (secuencias de escape)** dentro de una cadena de texto.

Control	Función	Explicación	Comando de Ejemplo
r""	Prefijo `r` (raw)	Se coloca la letra `r` justo antes de las comillas que delimitan el *string*. Esto asegura que las barras invertidas (\) sean tratadas como caracteres literales y no como comandos de escape.	`print(r"C:\Usuarios\Python\nueva_carpeta")`
Salida		La salida resultante es **C:\Usuarios\Python\nueva_carpeta**.	

---

3. Limpiar Pantalla de Consola (Control del Sistema)

Este control permite ejecutar comandos del sistema operativo (SO) para manipular la interfaz de la consola.

Módulo/ Comando	Propósito	Secuencia de Comandos	Explicación del Control
`from os import system`	Importación	Importa la función `system` del módulo `os` (Operating System).	Prepara el entorno para ejecutar comandos a nivel de SO.
`system("cls")`	Ejecución	Llama a la función `system` y le pasa el comando `"cls"`.	En sistemas Windows, `"cls"` es el comando para limpiar la pantalla de la consola.

---

4. Bucles Vacíos (Control de Estructura)

El comando **`pass`** es un control de estructura que se utiliza para crear un bucle (o cualquier bloque de código) que, aunque sintácticamente necesario, no realiza ninguna acción.

Comando	Propósito	Explicación Detallada	Comando de Ejemplo
`pass`	Bucle Vacío	Se utiliza como un *placeholder* o marcador de posición. Cuando la sintaxis de un bloque requiere un comando, pero el programador no desea que se ejecute nada (por ejemplo, mientras se está desarrollando o planeando la lógica).	`for i in range(5): pass`
Estructura	Bucle	En este ejemplo, el bucle `for` iterará 5 veces, pero en cada iteración, el comando `pass` asegura que no se ejecute ninguna instrucción dentro del bloque del bucle.	

18. Juegos

Juego 1: Adivina el número

Enunciado: El programa elige un número entre 1 y 10, y el jugador debe adivinarlo.

```
import random

numero = random.randint(1, 10)
intento = int(input("Adivina un número entre 1 y 10: "))

if intento == numero:
    print("¡Acertaste!")
else:
    print("Fallaste, el número era", numero)
```

Explicación:
import random -> importa la librería random.
random.randint(1, 10) -> genera un número aleatorio entre 1 y 10.
int(input(...)) -> pide un número y lo convierte en entero.
if intento == numero -> compara intento y número.
print(...) -> muestra mensajes en pantalla.

Juego 2: Piedra, papel o tijera

Enunciado: El usuario juega contra la computadora.

```
import random

opciones = ["piedra", "papel", "tijera"]
usuario = input("Elige piedra, papel o tijera: ")
pc = random.choice(opciones)

print("La computadora eligió:", pc)

if usuario == pc:
    print("Empate")
elif (usuario == "piedra" and pc == "tijera") or \
     (usuario == "papel" and pc == "piedra") or \
     (usuario == "tijera" and pc == "papel"):
    print("¡Ganaste!")
else:
    print("Perdiste")
```

Explicación:
opciones = [...] -> lista con jugadas posibles.
input(...) -> pide jugada al usuario.
random.choice(opciones) -> elige jugada al azar.
if, elif, else -> deciden el ganador.
print(...) -> muestra resultados.

Juego 3: Adivina la palabra

Enunciado: El jugador debe adivinar la palabra secreta con tres intentos.

```
palabra = "python"
intentos = 3

while intentos > 0:
    respuesta = input("Adivina la palabra secreta: ")
    if respuesta == palabra:
        print("¡Correcto!")
        break
    else:
        intentos -= 1
        print("Incorrecto. Te quedan", intentos, "intentos.")
```

Explicación:
palabra = "python" -> palabra secreta.
intentos = 3 -> número de intentos permitidos.
while intentos > 0 -> bucle que repite mientras queden intentos.
if respuesta == palabra -> verifica si acierta.
break -> termina el bucle si gana.
intentos -= 1 -> resta intentos.

Juego 4: Lanzar un dado

Enunciado: El programa simula el lanzamiento de un dado de seis caras.

```
import random

dado = random.randint(1, 6)
print("El dado muestra:", dado)
```

Explicación:
random.randint(1, 6) -> genera número aleatorio entre 1 y 6.
print(...) -> muestra el resultado.

Juego 5: Adivina el número con pistas

Enunciado: El jugador debe adivinar un número entre 1 y 20 con pistas de mayor o menor.

```
import random

numero = random.randint(1, 20)
intento = 0

while True:
    intento = int(input("Adivina un número entre 1 y 20: "))
    if intento == numero:
        print("¡Correcto!")
        break
    elif intento < numero:
        print("El número es mayor")
    else:
        print("El número es menor")
```

Explicación:
while True -> bucle infinito hasta acertar.
if intento == numero -> condición de acierto.
elif intento < numero -> pista de número mayor.
else -> pista de número menor.
break -> termina cuando acierta.

Juego 6: Cara o cruz

Enunciado: Simula lanzar una moneda.

```
import random

moneda = random.choice(["cara", "cruz"])
print("La moneda cayó en:", moneda)
```

Explicación:
random.choice([...]) -> elige cara o cruz al azar.
print(...) -> muestra el resultado.

Juego 7: Ahorcado simple

Enunciado: El jugador debe adivinar una palabra letra por letra.

```
palabra = "gato"
oculto = ["_"] * len(palabra)

while "_" in oculto:
    print(" ".join(oculto))
    letra = input("Adivina una letra: ")
    if letra in palabra:
        for i in range(len(palabra)):
            if palabra[i] == letra:
                oculto[i] = letra
    else:
        print("Letra incorrecta")

print("¡Ganaste! La palabra era", palabra)
```

Explicación:
oculto = ["_"] * len(palabra) -> lista de guiones bajos.
" ".join(oculto) -> muestra guiones separados por espacio.
for i in range(len(palabra)) -> recorre índices de palabra.
if palabra[i] == letra -> reemplaza guion por letra correcta.

Juego 8: Número par o impar

Enunciado: La computadora elige un número y el jugador adivina si es par o impar.

```
import random
```

```
numero = random.randint(1, 100)
print("Número:", numero)

respuesta = input("¿Es par o impar?: ")
if numero % 2 == 0 and respuesta == "par":
    print("Correcto")
elif numero % 2 != 0 and respuesta == "impar":
    print("Correcto")
else:
    print("Incorrecto")
```

Explicación:
numero % 2 == 0 -> verifica si es par.
input(...) -> pide respuesta al usuario.
if, elif, else -> evalúan la respuesta.

Juego 9: Adivina el número en intentos limitados

Enunciado: El jugador tiene 5 intentos para adivinar un número entre 1 y 50.

```
import random

numero = random.randint(1, 50)
intentos = 5

while intentos > 0:
    guess = int(input("Adivina un número: "))
    if guess == numero:
        print("¡Correcto!")
        break
    elif guess < numero:
        print("Demasiado bajo")
    else:
        print("Demasiado alto")
    intentos -= 1

if intentos == 0:
    print("Perdiste. El número era", numero)
```

Explicación:
intentos = 5 -> máximo de intentos.
while intentos > 0 -> bucle limitado.
elif guess < numero -> pista de número mayor.
else -> pista de número menor.

Juego 10: Juego de sumas

Enunciado: El programa da una suma aleatoria que el jugador debe resolver.

```
import random

a = random.randint(1, 10)
b = random.randint(1, 10)
respuesta = int(input(f"¿Cuánto es {a} + {b}?: "))
```

```
if respuesta == a + b:
    print("Correcto")
else:
    print("Incorrecto. La respuesta era", a + b)
```

Explicación:
f"..." -> f-string para insertar variables en el texto.
a + b -> calcula la suma.
if respuesta == a + b -> compara respuesta.

Juego 11: Adivina el color

Enunciado: La computadora elige un color de una lista y el jugador debe adivinarlo.

```
import random

colores = ["rojo", "azul", "verde", "amarillo"]
secreto = random.choice(colores)

eleccion = input("Adivina un color (rojo, azul, verde, amarillo): ")

if eleccion == secreto:
    print("¡Acertaste!")
else:
    print("Fallaste, el color era", secreto)
```

Explicación:
colores = [...] -> lista de opciones.
random.choice(colores) -> elige color al azar.
input(...) -> pide un color al jugador.

Juego 12: Suma de dados

Enunciado: El jugador lanza dos dados y gana si la suma es 7.

```
import random

dado1 = random.randint(1, 6)
dado2 = random.randint(1, 6)
suma = dado1 + dado2

print("Dado 1:", dado1, "Dado 2:", dado2, "Suma:", suma)

if suma == 7:
    print("¡Ganaste!")
else:
    print("Perdiste")
```

Explicación:
dado1, dado2 -> generan valores de dados.
suma = dado1 + dado2 -> calcula la suma.
if suma == 7 -> condición de victoria.

Juego 13: Cuenta atrás

Enunciado: Juego simple que cuenta de 5 a 1 antes de "¡Despegue!".

```
import time

for i in range(5, 0, -1):
    print(i)
    time.sleep(1)

print("¡Despegue!")
```

Explicación:
for i in range(5, 0, -1) -> bucle decreciente de 5 a 1.
time.sleep(1) -> pausa de un segundo.

Juego 14: Mayor de dos números

Enunciado: El jugador introduce dos números y el programa dice cuál es mayor.

```
a = int(input("Escribe el primer número: "))
b = int(input("Escribe el segundo número: "))

if a > b:
    print(a, "es mayor")
elif b > a:
    print(b, "es mayor")
else:
    print("Son iguales")
```

Explicación:
if a > b -> compara números.
elif b > a -> segunda comparación.
else -> caso de igualdad.

Juego 15: Tabla de multiplicar

Enunciado: El jugador introduce un número y el programa muestra su tabla de multiplicar.

```
n = int(input("Introduce un número: "))

for i in range(1, 11):
    print(n, "x", i, "=", n * i)
```

Explicación:
for i in range(1, 11) -> bucle de 1 a 10.
print(n, "x", i, "=", n * i) -> muestra cada multiplicación.

Juego 16: Adivina la letra

Enunciado: El jugador debe adivinar una letra secreta.

```
import random
```

```
letras = "abcdefg"
secreta = random.choice(letras)

eleccion = input("Adivina una letra (a-g): ")

if eleccion == secreta:
    print("¡Correcto!")
else:
    print("Incorrecto, la letra era", secreta)
```

Explicación:
letras = "abcdefg" -> cadena de opciones.
random.choice(letras) -> elige letra al azar.

Juego 17: Juego del conteo

Enunciado: El jugador escribe un número y el programa cuenta hasta él.

```
n = int(input("Escribe un número: "))

for i in range(1, n + 1):
    print(i)
```

Explicación:
for i in range(1, n + 1) -> bucle que cuenta de 1 hasta n.

Juego 18: Número secreto con rango

Enunciado: Adivinar un número entre 1 y 100, el programa da pistas.

```
import random

secreto = random.randint(1, 100)

while True:
    intento = int(input("Adivina el número: "))
    if intento == secreto:
        print("¡Correcto!")
        break
    elif intento < secreto:
        print("Muy bajo")
    else:
        print("Muy alto")
```

Explicación:
while True -> bucle infinito hasta acertar.
if intento == secreto -> condición de acierto.
elif intento < secreto -> pista de bajo.
else -> pista de alto.

Juego 19: Juego del cuadrado

Enunciado: El jugador debe responder el cuadrado de un número.

```
import random

n = random.randint(1, 10)
respuesta = int(input(f"¿Cuál es el cuadrado de {n}?: "))

if respuesta == n ** 2:
    print("Correcto")
else:
    print("Incorrecto, era", n ** 2)
```

Explicación:

n ** 2 -> calcula el cuadrado del número.

if respuesta == n ** 2 -> compara con la respuesta.

Juego 20: Número mayor de tres

Enunciado: El jugador introduce tres números y el programa dice cuál es mayor.

```
a = int(input("Primer número: "))
b = int(input("Segundo número: "))
c = int(input("Tercer número: "))

if a > b and a > c:
    print(a, "es el mayor")
elif b > a and b > c:
    print(b, "es el mayor")
else:
    print(c, "es el mayor")
```

Explicación:

if a > b and a > c -> comprueba si a es mayor.

elif b > a and b > c -> comprueba si b es mayor.

else -> en cualquier otro caso, c es el mayor.

¿Quieres que te prepare también una **versión más avanzada** de estos juegos (con más funciones, niveles o dificultad), o prefieres quedarte con estos ejemplos básicos y claros?

Anexo

1. Caracteres especiales

1 Saltos de línea y espaciado

Carácter	Significado	Ejemplo	Salida
`\n`	Nueva línea	`print("Hola\nMundo")`	HolaMundo
`\t`	Tabulación horizontal	`print("Col1\tCol2\tCol3")`	Col1 Col2 Col3
`\r`	Retorno de carro (inicio de línea)	`print("12345\rAB")`	AB345
`\v`	Tabulación vertical	`print("Linea1\vLinea2")`	Linea1Linea2 (vertical)
`\f`	Salto de página	`print("Linea1\fLinea2")`	Linea1Linea2 (salto de página)

---

2 Comillas y barra invertida

Carácter	Significado	Ejemplo	Salida
`\'`	Comilla simple dentro de comillas simples	`print('It\'s Python')`	It's Python
`\"`	Comilla doble dentro de comillas dobles	`print("Ella dijo: \"Hola\"")`	Ella dijo: "Hola"
`\\`	Barra invertida literal	`print("C:\\Usuarios\\Python")`	C:\Usuarios\Python

---

3 Unicode y valores hexadecimales

Carácter	Significado	Ejemplo	Salida
`\uXXXX`	Carácter Unicode (4 dígitos hex)	`print("\u2764")`	❤
`\UXXXXXXXX`	Carácter Unicode (8 dígitos hex)	`print("\U0001F600")`	😀
`\xXX`	Carácter hexadecimal	`print("\x41")`	A

---

4 Sonidos y control de texto

Carácter	Significado	Ejemplo	Salida
`\a`	Sonido de alerta	`print("\a")`	(campana/suena alerta)
`\b`	Retroceso (borra carácter anterior)	`print("Hola\bMundo")`	HolMundo
`\0`	Carácter nulo	`print("A\0B")`	AB (con carácter nulo entre medio, no visible)

5️⃣ Strings crudos (raw strings)

Si quieres **ignorar caracteres especiales**, pon `r` antes de las comillas:

```
print(r"C:\Usuarios\Python\nueva_carpeta")
```

Salida:

```
C:\Usuarios\Python\nueva_carpeta
```

---

2. Limpiar pantalla de consola

from os import system

system(“cls”)

3. Bucles vacios

for i in range(5):

    pass #bucle vacio

Epílogo

Con la última página de este libro llegamos al cierre de un recorrido que comenzó con los fundamentos más sencillos de Python y avanzó hacia problemas de mayor complejidad, diseñados para desafiar la lógica, la creatividad y la capacidad de resolución del lector. Cada ejercicio ha sido una oportunidad para poner en práctica los conceptos que sustentan la programación, y a la vez una invitación a pensar, experimentar y construir soluciones propias.

Este libro de ejercicios no debe entenderse como un final, sino como un punto de partida. El verdadero aprendizaje de la programación se consolida en la práctica continua, en la curiosidad por explorar nuevos retos y en la perseverancia para resolver problemas cada vez más exigentes. Python es un lenguaje versátil y poderoso, pero lo más valioso que el lector puede llevarse de esta experiencia no es solo el dominio de una sintaxis o de un conjunto de funciones, sino la forma de pensar que la programación desarrolla: analítica, ordenada y creativa.

Quienes han llegado hasta aquí cuentan ahora con una base sólida para continuar su propio camino en el mundo de la programación. La práctica de los ejercicios, sumada a la consulta del manual explicativo que acompaña a esta obra, constituye una herramienta de aprendizaje integral que busca no solo enseñar a programar, sino también fomentar la autonomía y la confianza para enfrentar nuevos desafíos.

Esperamos que este libro haya sido, más que una colección de problemas, un compañero de viaje que motive a seguir explorando. Que cada ejercicio resuelto sea la semilla de futuros proyectos, ideas y desarrollos que den sentido a lo aprendido.

El camino de la programación es infinito. Este epílogo no es un cierre definitivo, sino una invitación a continuar.